2014-09

Application transparent HTTP over a disruption tolerant Smart-Net

Alt, Lance

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

APPLICATION TRANSPARENT HTTP OVER A DISRUPTION TOLERANT SMARTNET

by

Lance Alt

September 2014

Thesis Co-Advisors: Geoffrey G. Xie
Justin P. Rohrer

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	09-26-2014	Master's Thesis 10-21-2013 to 09-26-2014

4. TITLE AND SUBTITLE

APPLICATION TRANSPARENT HTTP OVER A DISRUPTION TOLERANT SMARTNET

5. FUNDING NUMBERS

6. AUTHOR(S)

Lance Alt

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of the Navy

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

This research explores methods to increase the performance of HTTP traffic when operating on a network that is prone to disruptions. The SmartNet architecture is presented as an open and extensible software framework for experimenting with and deploying application-transparent network optimization solutions, including the incorporation of the disruption tolerant networking (DTN) and split TCP (SplitTCP) technologies into an IP network. The architecture fashions a plugin-based system architecture where each plugin implements a small set of application or transport protocol specific network adaptations that can be chained with other plugins to form a packet processing pipeline. The SmartNet framework is implemented along with plugins to route packets through native-IP, the Bundle Protocol, or SplitTCP. Performance of the SmartNet is measured under five network disruption patterns and five link speeds. The results conclude that HTTP performance can be increased by using the SmartNet to transparently route packets over the DTN bundle protocol or SplitTCP when the network is prone to disruptions.

14. SUBJECT TERM	15. NUMBER OF PAGES
Computer Networking, Disruption Tolerant Networks, SplitTCP, Network Optimization, Middlebox, Overlay Networks	93
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

APPLICATION TRANSPARENT HTTP OVER A DISRUPTION TOLERANT SMARTNET

Lance Alt
Lieutenant, United States Navy
B. S., University of Texas at Arlington, 2006

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2014

Author: Lance Alt

Approved by: Dr. Geoffrey G. Xie
Thesis Co-Advisor

Dr. Justin P. Rohrer
Thesis Co-Advisor

Dr. Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This research explores methods to increase the performance of HTTP traffic when operating on a network that is prone to disruptions. The SmartNet architecture is presented as an open and extensible software framework for experimenting with and deploying application-transparent network optimization solutions, including the incorporation of the disruption tolerant networking (DTN) and split TCP (SplitTCP) technologies into an IP network. The architecture fashions a plugin-based system architecture where each plugin implements a small set of application or transport protocol specific network adaptations that can be chained with other plugins to form a packet processing pipeline. The SmartNet framework is implemented along with plugins to route packets through native-IP, the Bundle Protocol, or SplitTCP. Performance of the SmartNet is measured under five network disruption patterns and five link speeds. The results conclude that HTTP performance can be increased by using the SmartNet to transparently route packets over the DTN bundle protocol or SplitTCP when the network is prone to disruptions.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

Appendices

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

API application programming interface

ASCII American Standard Code for Information Interchange

BP Bundle Protocol

BPA bundle protocol agent

CLA convergence layer adapters

CPU central processing unit

DTN disruption tolerant network

HTTP Hypertext Transport Protocol

ICMP Internet Control Message Protocol

IDS intrusion detection system

IP Internet Protocol

LWIP Lightweight Internet Protocol

MTU maximum transmission unit

NASA National Aeronotics and Space Administration

OS operating system

OSI Open Systems Interconnection

RAM random access memory

RTT round trip time

SDN software defined networking

SIP Session Initiation Protocol

TCP Transmission Control Protocol

UDP User Datagram Protocol

VoIP Voice over IP

CHAPTER 1:
Introduction

During its inception, the Internet was based around the Transmission Control Protocol (TCP) and Internet Protocol (IP) for reliable end-to-end routing of packets. Both protocols have been resilient to changing requirements and are able to adapt to many different situations. TCP and IP were designed during a time where network devices were commonly connected via hardwire connections with low loss rate and latency. More recently, mobile devices and wireless networking have expanded the Internet to include links that have high latency and are prone to disruption. Over time, TCP and IP have slowly evolved in an attempt to maximize performance on disrupted networks; however, the standard TCP/IP stack is still less than ideal when operating on links with higher loss rates and latency[1]. With the rapid growth of the Internet, reliance on computer networks has become a critical part of society, demanding fast and reliable connections even on networks prone to disruption. This research explores the implications of running TCP on disrupted networks and methods to increase performance on these networks.

TCP was developed using an end-to-end connectivity model. In this model, the end hosts are responsible for all aspects of the connection including detecting lost packets, packet retransmission, flow control, and congestion control. This works well in large networks, such as the Internet, because the internal routers do not care about connection details and can focus on forwarding single packets at a fast rate. When loss is low this model is adequate; however, on networks with a high loss rate, the end-to-end model becomes burdensome not only on the end hosts, but on the network as a whole.

Figure 1.1 shows a typical network in which a TCP connection might be established. In this situation, the packets travel along a single path (shown in red) between the client and server end hosts. The network depicted also has a link between R4 and R6 that is prone to disruption. TCP ensures reliability by requiring the destination host to transmit a positive acknowledgment to the sending host. In this scenario, when the client transmits a packet to the server, it will wait for the acknowledgment. If the packet is lost over the disrupted link, the client will wait for a predetermined amount of time for the acknowledgment, and then

attempt to retransmit the packet. Even though the original packet made it all the way to R4, the retransmitted packet in its entirety must traverse through R1 and R2 again, causing extra network strain on those routers and the networks they service. The strain on the network is compounded with the addition of multiple disrupted links along the end-to-end path. If several links are disrupted independently the probability of having a complete end-to-end path decreases significantly. In such cases, it is likely that TCP will terminate the connection after several failed retransmissions. Additionally, certain TCP connection states, such as the sending of the initial SYN during connection establishment, are particularly vulnerable to disruption. In the case of a lost SYN, most TCP implementations will give up on establishing a connection after fewer retransmission attempts.

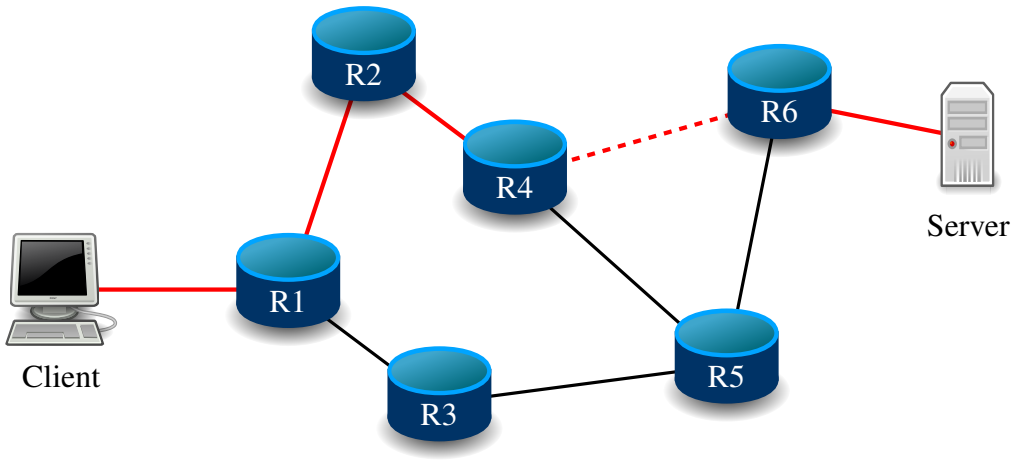


Figure 1.1: A typical network configuration with a disrupted connection (dashed). An end-to-end TCP path is shown in red.

The alternative to the end-to-end connectivity model is a hop-by-hop model. In a hop-by-hop model, each router is responsible for ensuring a packet successfully arrives at the next hop. If a packet is lost between the R4 and R6 routers, the R4 router keeps a copy of the packet and will retransmit it after a certain timeout period. Using a hop-by-hop model eliminates the need for the client to retransmit the packet again through R1 and R2; however, it places additional burden on R4, which needs to maintain state for each packet passing through it.

Disruption tolerant networking (DTN) is a field within the computer networking that studies the effect disruption has on networks and ways to mitigate the effect of the disruptions. Since TCP uses end-to-end connections, several alternative protocols have been developed to support hop-by-hop connections. The first and most prominent DTN protocol is the Bundle Protocol (BP) as specified in RFC5050[2]. The BP operates by encapsulating data inside a bundle, which is then forwarded through a BP-capable network. At each hop within the network, custody of the bundle is transferred once verification has been made that it arrived at the next hop successfully.

Another approach to supporting hop-by-hop communication is SplitTCP[3]. SplitTCP aims to adapt TCP to support hop-by-hop connections while maintaining compatibility with existing TCP/IP network infrastructure. Each SplitTCP router within a end-to-end connection intercepts all TCP packets flowing though it. Packets that require acknowledgment are acknowledged by the SplitTCP router and the router becomes responsible for ensuring the reliable transport of the packet from that point forward. A chain of SplitTCP routers will continue this process, each taking custody of the packet, in effect creating a hop-by-hop TCP connection. Like all hop-by-hop solutions, SplitTCP imposes extra processing and storage requirements on the SplitTCP routers.

The research invested in various DTN technologies has resulted in several successful implementations such as the National Aeronotics and Space Administration (NASA) deep space network[4]. While DTN technology has made an impact in special case scenarios, it has struggled to gain acceptance on the wider Internet. One major hold-back of widespread adoption of this technology is that most DTN solutions require significant changes to the application, replacing TCP/IP as the primary communication protocol. The predominant approach to integrating DTN networks follows a vertical overlay model. In the case of the BP, it is either IP-over-BP or BP-over-IP. This layered approach is simple to design and implement, because neither data translation or the tracking of application states are needed inside the network. However, it not only introduces extra encapsulation overhead, but more importantly, imposes least-common denominator semantics when moving data across network boundaries, and as such, may severely degrade the performance of many applications originally designed to work over end-to-end IP connectivity. There is a prevailing perception that the DTN technology is not plug-and-play and existing applications

must be retrofitted to use DTN. This might explain the surprisingly limited deployment of DTN, even though DTN has been repeatedly demonstrated to be beneficial in many scenarios involving challenged networks. For these reasons, any effort to gain widespread DTN acceptance would require compatibility with TCP/IP.

This research proposes a new architecture for general application-transparent network optimization called SmartNet. SmartNet is designed to replace selected routers within a network where dynamic network optimization would be beneficial. The SmartNet is designed to be flexible, using a plugin-based architecture where each plugin performs specific network optimization. The plugins are chained together into a pipeline through which packets are routed. Since the SmartNet is software based and configurable, it can make intelligent decisions about which optimization to perform based on the network state or details of a particular TCP connection.

For example, in Figure 1.2, R4 and R6 have been replaced with SmartNets. Since it is known that the link between these two routers was subject to disruption, the SmartNets can be used to optimize packet flow over this hop. The SmartNet could be used to implement either the BP or SplitTCP over this disrupted link, requiring no changes to either the client or server, or any of the other network devices. The dynamic nature of the SmartNet allows for arbitrarily complex decision points. For example, the SmartNet configuration might determine that real-time traffic, such as Voice over IP (VoIP), should be rerouted via R5 so that it arrives as quickly as possible while simultaneously buffering bulk traffic and transferring it only when the disrupted link becomes available again.

This research explores the use of SmartNet technology to implement DTN protocols in an application transparent way and provides the following three primary contributions:

1. Design and implement a general framework for application-transparent network optimization, called SmartNet. The open design of SmartNet allows easy implementation of new features and unlimited customization.
2. Use the SmartNet to transparently use both the BP and SplitTCP as a means of mitigating the effects of a disrupted network. Transparently using DTN technology requires no changes to existing applications.

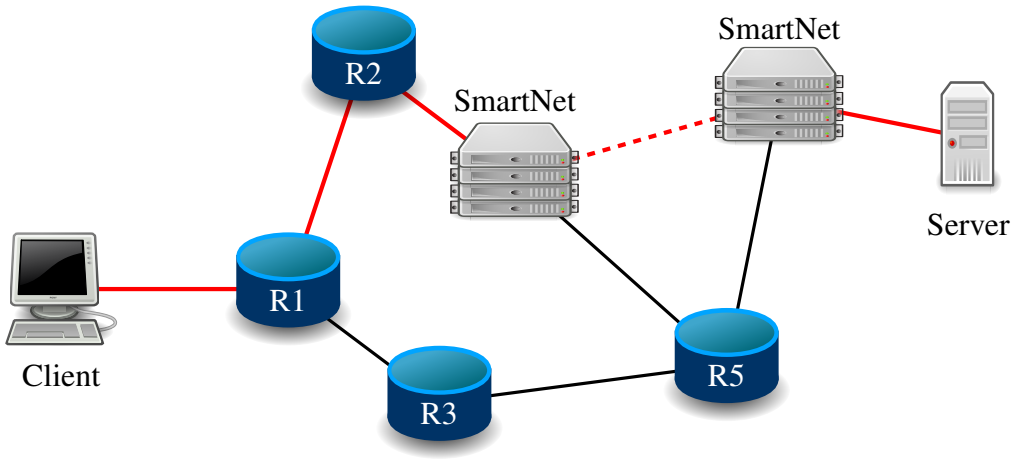


Figure 1.2: A typical network configurations using SmartNet to mitigate the effects of the disrupted connection (dashed). The end-to-end TCP path is shown in red.

3. Measure the performance of the SmartNet in both disrupted and non-disrupted scenarios and compare to the performance to that of a normal TCP/IP network.

Chapter 2 explores several other architectures as possible candidates [3], [5]–[10] for seamless integration of DTN technology in to existing TCP/IP networks. Many of the existing architectures provide some of the required functionality, but none satisfy all the criteria needed to transparently support disrupted networks.

Therefore, in Chapter 3, a general framework for application-transparent network optimization, SmartNet, is designed which can be used to alleviate the effects of network disruption or degradation. The SmartNet acts as a network router that can dynamically alter the flow of network traffic over multiple independent connections. The SmartNet design is open and extensible, using a plugin-based system architecture where each plugin implements a small set of application or transport protocol specific network adaptation requirements and can be chained with other plugins to form a packet processing pipeline.

Chapter 4 describes a specific SmartNet implementation, NpsGate. The NpsGate core is implemented along with several plugins each providing specific network optimization functionality. Plugin pipelines for native IP, SplitTCP, and DTN are developed for evaluating the performance of the NpsGate implementation.

In Chapter 5 NpsGate is deployed to test the SmartNet architecture. First, `iperf` is used to evaluate the effect the SmartNet has on network throughput including the maximum throughput NpsGate supports. Second, a HTTP download scenario is tested against a variety of link speeds and disruption patterns to quantify the performance benefits of using the SmartNet.

Final conclusions about the SmartNet design and performance are discussed in Chapter 6. Summaries of the results found in Chapter 5 are presented and discussed along with analysis of the effectiveness of the SmartNet. Weaknesses and limitations of the SmartNet design are reviewed along with goals of future work on the system.

CHAPTER 2:
Background

The problems surrounding the use of TCP/IP over disrupted networks are well known [1]. TCP provides the majority of reliable end-to-end inter-process communication on the Internet. It is designed in such a way as to tolerate lost packets, network congestion, and high latencies. TCP accomplishes this by requiring a positive acknowledgment for each packet received. If an acknowledgment is not received within a predetermined amount of time, the packet is retransmitted. The TCP model is acceptable on networks where packet loss is a rarity; however, on a network prone to disruption, retransmission can cause an unnecessary increase in network traffic. Figure 2.1 shows an example where, after a network disruption, the sender must retransmit the entire data packet even though the receiver properly received the packet.

In addition, the sender has no way of knowing whether the network is in a disrupted state. The sender may try to retransmit the packet while the network is still in a disrupted state. Eventually, after a certain number of retransmissions (typically three), the sender will assume that the connection has been terminated and will close the connection. The application must then re-initiate the connection, which is often a lengthy process, and typically requires user interaction. In an application such as a web browser, disconnection often results in the complete loss of a partially downloaded file, requiring the application to start the download again from scratch. This can result in hundreds of megabytes of duplicated transmissions and can make it nearly impossible to complete large file downloads if the connection is often disrupted.

2.1 Disruption Characteristics

Since this research focuses on operation over disrupted networks, it is important to fully understand the characteristics of a disruption and under what circumstances they occur. Disruption is categorized into three distinct categories: high latency/low data rate, disconnection, and long queuing times [9].

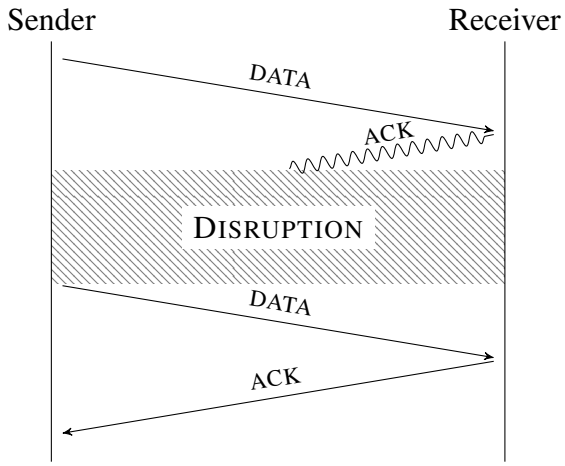


Figure 2.1: Example of TCP retransmission on a disrupted network. Even though the data arrived at the receiver, the acknowledgment was lost during a period of disruption requiring retransmission of the entire data packet.

2.1.1 High Latency/Low Data Rate

The high latency/low data rate condition occurs in heterogeneous networks where certain paths have higher latencies or lower data rates compared to the rest of the network. The example given in [9] is that of an underwater acoustic network where data rates of 10 Kbit/s and latencies of one to two seconds are common. Asymmetric links, found commonly in satellite connections, where the uplink data rate is significantly less than the downlink are another example of a situation where high latency/low data rate disruptions are possible.

2.1.2 Disconnection

Disconnection occurs when the physical medium between two network segments becomes disrupted such that no information can be transmitted. This type of disruption often occurs in wireless environments and can either be predictable, such as satellite passes, or sporadic, such as nodes moving out of communication range[9]. Disconnection typically occurs as a result of motion by either of the communicating nodes; however, it can also occur due to action by a third party. For example, a microwave connection could be disconnected when a low flying aircraft passes through the two nodes. The duration of a disconnection can be as short as a second, as in the microwave example, or could last hours or more, such as when a planetary satellite becomes occluded by the planet it is orbiting.

2.1.3 Long Queuing Times

In conventional multi-hop paths, queuing delays dominate propagation and transmission delays[9]. Even so, the delays are typically short, rarely exceeding a second. When processing packets, routers will typically drop a packet if the next hop is not instantly reachable. Queuing times while operating on a disrupted network, however, may be much longer. In addition, source-initiated retransmissions on a disrupted network may be expensive in both bandwidth and time.

Most current routing implementations have a default policy of dropping new packets when the queue is full. During a period of disruption, queuing times may increase significantly, resulting in many dropped packets. A disruption tolerant router needs the added capability to handle large queue sizes while the network is disrupted.

2.2 DTN Architecture

In 2003, an architecture for a DTN was proposed in [9]. The architecture focuses around designing a general purpose overlay architecture operating above existing protocol stacks. The architecture is agnostic to the underlying protocol but is typically used with the TCP/IP stack and operates as an application overlay. Figure 2.2 shows an example protocol stack utilizing the OSI Network model for a DTN-enabled application. This architecture was eventually used as the basis for the Bundle Protocol as specified in RFC5050 [2].

In this DTN architecture, applications would need to be specifically written to use the overlay. This constraint is undesirable in general because the task of rewriting numerous large software projects to take advantage of the DTN architecture is both time and cost prohibitive.

The BP, defined in RFC5050[2], is the standard for DTN based communication. There are several independent implementations of the specification [7], [11], [12]. The BP provides hop-by-hop reliability and buffering capable of coping with intermittent connectivity and taking advantage of opportunistic connectivity. As with the architecture in [9], BP operates above the transport layer and requires applications to be specifically written to use the protocol. Since the BP is standardized and has multiple independent implementations, it is an ideal choice to use in a SmartNet system to provide reliability over distrupted net-

works. One key obstacle to overcome this is to provide application transparency while operating over the BP. To achieve this, the SmartNet must provide some type of middlebox functionality to translate standard TCP/IP connections to operate seamlessly over the BP.

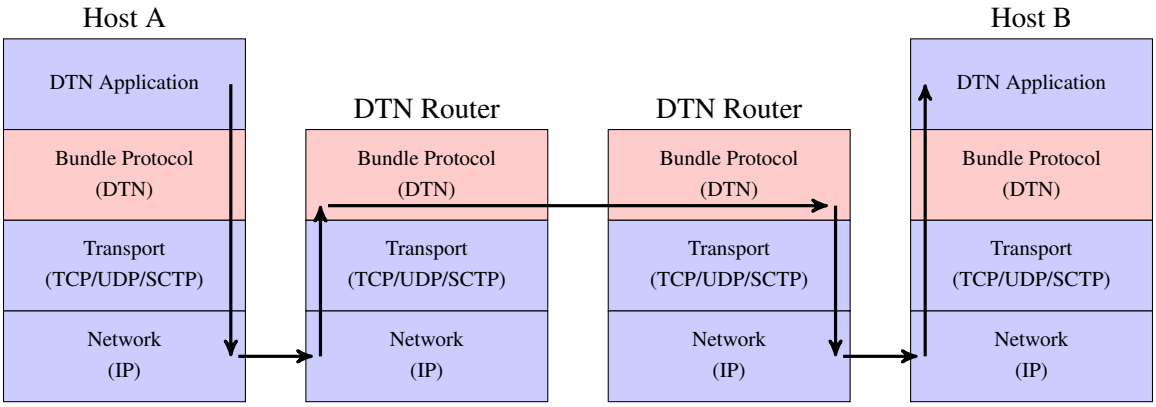


Figure 2.2: DTN protocols operate using an overlay approach. Various transport and network layer protocols can be used as the foundation for the Bundle Protocol, shown in red. While this provides the ability to operate on existing TCP/IP networks, it requires specific application support to take advantage of the DTN layer.

2.3 Middlebox Middleware

Prior work has focused on the vast array of middleboxes found in large network architectures. Middleboxes are specialized network appliances such as proxies, firewalls, or intrusion detection systems (IDSs) that perform a specific functionality. These middleboxes are often closed solutions, lacking the ability to extend or customize their capability [13]. In [5], the authors design an architecture called CoMb, which is designed to consolidate middlebox deployments. CoMb consists of three primary components: classifier, policy enforcement, and middlebox applications as seen in Figure 2.3.

The classifier and policy enforcement layers represent the new contributions by this paper with the middlebox applications remaining unchanged from their current implementation. The classifier layer serves to consolidate the process of decoding and classifying packets based on fields in the TCP and IP headers. Once the packets are classified, they are sent to the policy enforcement layer. This layer is responsible for taking the classified packets and sending them to the appropriate middlebox application based on a set of policy rules.

The CoMb architecture presents several components such as an extensible application programming interface (API) and the use of queues between different layers that would be beneficial to the SmartNet architecture. However, CoMb fails to address several issues pertinent to a successful SmartNet deployment. First, CoMb does not provide an interface that allows modules at different layers to communicate with each other. This capability is critical to the SmartNet, since modules at all levels must be able to adapt to changing network conditions. The CoMb architecture is designed specifically around middleboxes and as such, lacks the scope needed for a SmartNet. A SmartNet architecture must be able to operate on packets at all levels both before and after processing by a middlebox and have the ability to interface directly with the host operating system (OS) network interfaces.

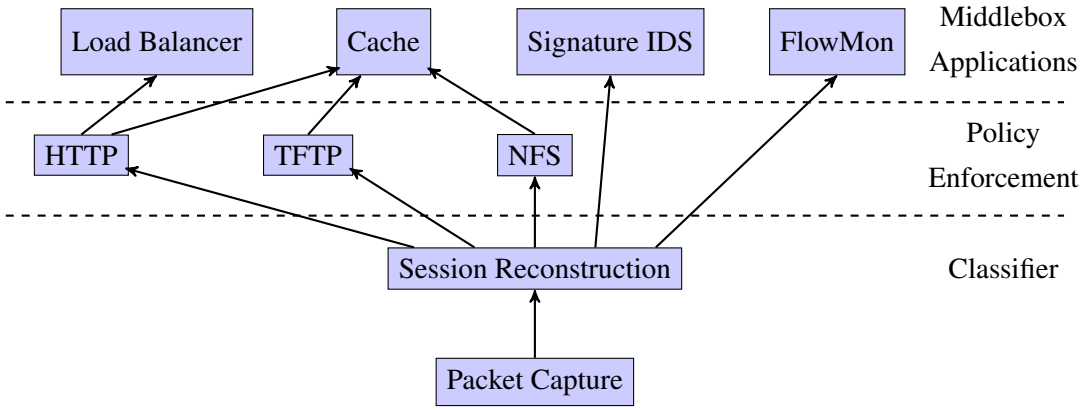


Figure 2.3: CoMb middlebox implementation. The CoMb architecture implements a three-tiered system to aid in middlebox deployment. A packet is first classified, then sent policy enforcement modules before ending up a existing middlebox solutions, after [5].

Another software defined networking (SDN)-based solution is presented in [6] as the SIMPLE system. SIMPLE is designed as a policy enforcement layer that can efficiently steer traffic to middlebox applications. SIMPLE uses tags and tunnels to eliminate the need for individual routers to make policy-based routing decisions for individual packets. As with the CoMb architecture, SIMPLE is designed to operate as a shell around existing middlebox applications and does not cover the scope needed for a SmartNet. In addition, the SIMPLE architecture specifically does not handle unanticipated link failures, a factor that is a key requirement for the SmartNet.

2.4 ClickOS

The architecture described in [8] is another example of a SDN approach to the problem of dynamic network processing. It works in a modular fashion by utilizing the Xen hypervisor to create multiple ClickOS instances. Each ClickOS is a combination of MiniOS, a basic OS provided by Xen, and Click, a modular router subsystem. Figure 2.4 graphically shows the ClickOS architecture. Within the ClickOS, a configuration file specifies a graph of connected Click elements. Each Click element performs a specific function such as packet classification, traffic shaping, or modification of header fields.

The ClickOS architecture is significantly closer to achieving an adequate SmartNet solution than the previously discussed systems. ClickOS is flexible due to its Click-based plugin system and requires no changes to the end-user application or OS. In addition, it promotes an open API, in effect turning the typical middlebox solution from a blackbox into a toolbox for meeting a network's specific needs, including new requirements for network adaption.

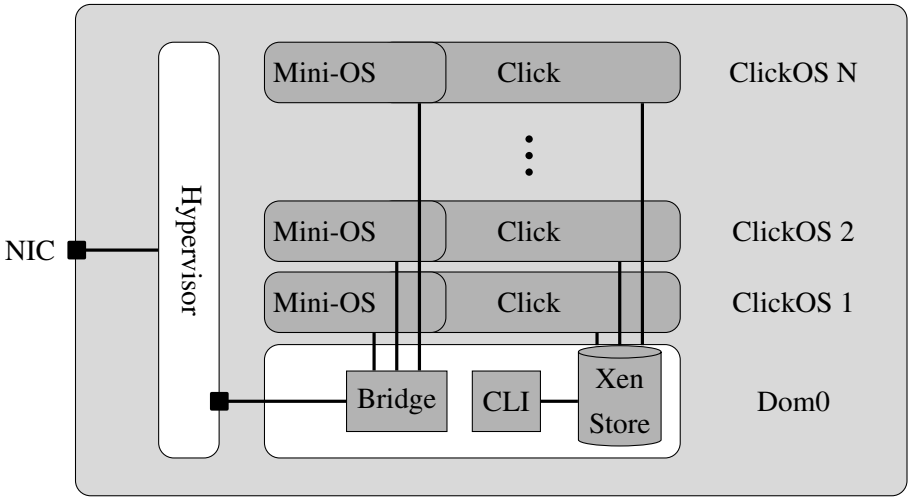


Figure 2.4: An overview of the ClickOS architecture. A hypervisor is used to create multiple Mini-OS instances, each of which contains a Click modular router, after [8].

ClickOS is closer to an ideal SmartNet architecture; however, it suffers from some undesirable features due to its reliance on the Click modular router. Click is capable of running in user-mode, but its primary target is running in kernel-mode. This implies a much lower-level interaction, requiring OS specific modules and interfacing directly with kernel level

data structures. In addition, running in kernel-mode places extraneous security requirements on the code since a single faulty module could bring down the entire OS. Running in kernel mode does bring performance benefits; however, any performance losses by running in user-mode are negligible with respect to the network disruptions being mitigated.

Click also suffers from a complex configuration scheme that requires the network administrator to understand the interworkings of various protocols to build a complete plugin pipeline. Click trades ease of use for efficiency by requiring the configuration to work with low-level abstractions such as raw packets.

For example, stripping a protocol header from a packet is a common task when classifying packets. In the Click system, the task of stripping the Ethernet frame header requires the network administrator to know the inner details of the protocol. Figure 2.5 shows an example configuration using Click that strips an Ethernet frame from a raw packet. An Ethernet frame header is normally 14 bytes, hence the `Strip(14)` line; however, the frame could include the 802.1Q tag [14], in which case, the byte at offset 12 must be `0x8100` and 18 bytes must be stripped. Care must be taken by the network administrator to ensure that the Click configuration handles all possible edge cases. This problem becomes significantly harder moving up in the Open Systems Interconnection (OSI) network layers. For example, both IP and TCP support optional headers which may result in a variable header size. Each possible combination of supported headers must be captured in the Click configuration. The SmartNet architecture should be designed to allow complete customization without resorting to the low-level details found in a Click configuration.

```
cl :: Classifier(12/8100, -);
cl[0] -> Strip(18);
cl[1] -> Strip(14);
```

Figure 2.5: Stripping an Ethernet frame in Click

2.5 DTN-centric Solutions

In 2013, [10] describes an IP-*cum*-DTN architecture, combining both a traditional IP network with a DTN. A visual representation of the IP-*cum*-DTN architecture designed in

[10] is in Figure 2.6. Packets traverse through a pair of gateway routers connected to each other via IP and a DTN protocol. The gateway routers, upon detection of a disruption on the IP network, will reroute packets over the DTN. Once the packets arrive on the receiving end, they are translated back to the IP before forwarding on to the destination host. This rerouting happens transparently to the end hosts, thereby requiring no changes to the application to take advantage of the DTN. The authors of [10] were successful in implementing the described architecture on a testbed composed of commodity hardware. Their testbed was capable of dynamically routing both Internet Control Message Protocol (ICMP) and Session Initiation Protocol (SIP) protocols over DTN without modification to the applications.

While the architecture described in [10] was successfully implemented for ICMP and SIP, these protocols are relatively simple and do not cover all the necessary functionality needed to generalize to all commonly used protocols. The most significant unresolved issue is the handling of of connection-oriented protocols, such as TCP. Both ICMP and SIP are connection-less protocols meaning each packet operates independently and single lost or out of order packet does not disrupt the flow of communication between hosts. On the other hand, connection oriented protocols consist of a stream of data encapsulated in multiple packets, requiring the endpoints to maintain connection state. For SmartNet to be effective, it will need to replicate some of this state, which can be a challenge if the network is disrupted during the connection setup phase (when state is synchronized between the endpoints). SmartNet must ensure that this state is not lost as a result of disruption. This research aims to extend the architecture in [10] to improve HTTP reliability and performance in challenged network environments.

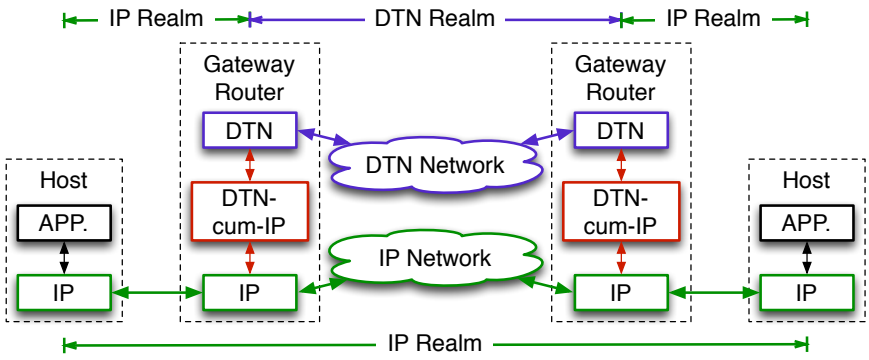


Figure 2.6: IP-*cum*-DTN layer architecture, from [10]

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Design Considerations

As seen in Chapter 2, several efforts have been made to implement middleware applications that operate on packets in a way that is transparent to the end-user application. There have also been several projects devoted to the design and implementation of a DTN and associated protocols. Some of these individual efforts have resulted in successful implementations that are used in production environments. This research focuses on the overarching goal of developing a single solution that combine both application transparent middleware and a DTN architecture. As a stepping stone, this research concentrates only on HTTP-based applications, with the intent that once a HTTP solution is developed, the lessons learned will provide the required insights for extending support to a wider range of applications.

3.1 SmartNet Requirements

The overall objective of this research is to design a solution that can be seamlessly implemented into existing networks rather than an independent solution requiring significant changes to established network ideas. To minimize transitional costs, the following requirements must be addressed in the SmartNet solution:

Use existing IP networks. Since the IP is deeply embedded in all Internet-based communications, the SmartNet must continue to operate over existing IP networks, and more importantly, use a DTN solution only when an IP connection is degraded.

Require no changes to existing applications. There are thousands of applications that use HTTP and rewriting all of these applications to support the SmartNet is cost- and time-prohibitive. Therefore, the SmartNet must operate transparently to these applications.

Have an extensible API. While this research focuses on HTTP, the SmartNet architecture should provide an extensible API that allows adaptation to other applications and protocols.

Allow programmable adaptability. The SmartNet is designed to operate in on dynamic networks with rapidly changing conditions. It must be flexible enough to detect abnormal conditions and adapt its behavior accordingly without human intervention.

Achieve high performance. The SmartNet must be able to operate at a performance level such that it does not cause significant delays on the network. Performance on high-speed low-latency networks, however, is not the target. This research focuses on networks where disruption is typical.

Easy to setup and configure. Key to the adaptation of any new technology is that it must be easy to setup and configure. The SmartNet should allow network administrators to easily configure basic functionality yet be it should be robust enough to allow for the configuration of complex capabilities.

3.2 SmartNet Design

Chapter 2 identified several competing architectures designed to support dynamic network adaptation; however, each of them has inherent pitfalls that underscore the need for a new architecture specifically designed to be application transparent while operating on a disrupted network. This new architecture is novel in that it combines several key capabilities: extensible plugin architecture, zero-copy packet processing, and inter-plugin communication.

3.2.1 Extensible Plugin Architecture

In order to facilitate an extensible architecture that can accommodate a wide variety of functionality, the SmartNet is designed to use plugins as building blocks. Plugins are configured to form packet processing pipelines. Input plugins interface with external sources, such as the OS or other user-space program, and selectively intercept and move new packets into the SmartNet. Processing plugins form the middle of the plugin pipeline and process packets received from input plugins or other processing plugins. Output plugins receive packets from processing plugins and redirect them to modules external to the SmartNet.

Each plugin is designed with a standardized external interface such that the output of any particular plugin can be designated as the input to any other plugin. The SmartNet manages the packet flow between a pair of plugins through the use of a packet queue, allowing individual plugins to operate asynchronously. Packet queues also allow flexibility when

utilizing plugins that may take a non-trivial amount time to process each packet. The queue will allow the slow plugin's predecessor to continue to process packets without waiting for the slower plugin to finish. In addition, the predecessor can query the queue size of the destination plugin, and redirect some or all of the packets to an alternate plugin if the queue is too long. Figure 3.1 illistrates the SmartNet plugin architecture.

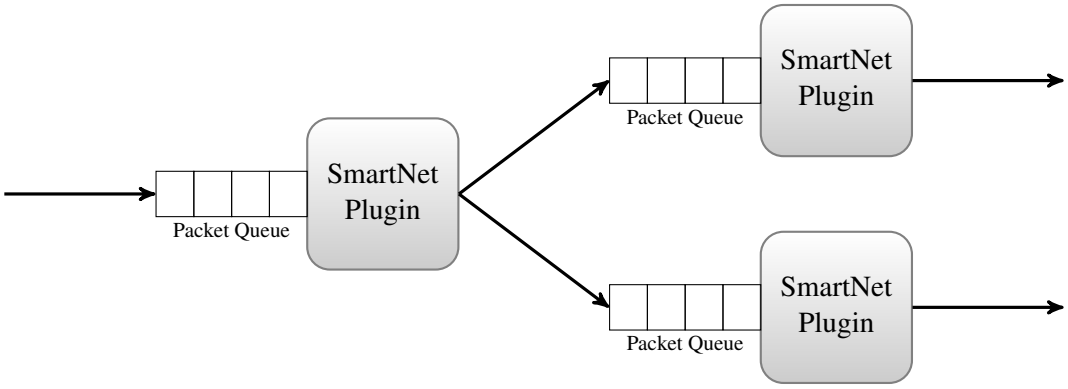


Figure 3.1: SmartNet plugin architecture. Each plugin contains an input queue where packets arrive. After processing is completed, the plugin forwards the packet to another plugin.

3.2.2 Zero-Copy Packet Processing

The basic data unit between plugins is a single IP packet. Operation at the packet level, rather than higher constructs such as streams, allows SmartNet plugins unlimited flexibility while handling packets. SmartNet is designed using a zero-copy method for passing packets between plugins. Each packet is managed by a shared packet storage unit and packets are passed between plugins as references. Each packet can be processed by a maximum of one plugin at any given time, allowing plugins the capability to modify the packet as part of their processing.

3.2.3 Inter-plugin Communication

To facilitate adaptability to a variety of network conditions, plugins within a pipeline must be able to communicate with each other. For example, an output plugin called IPOutput takes packets from its input queue and passes the packet to the host OS for transmission. There could be a situation where, for some reason, the IP route to the destination is down due to disruption. In this case, the IPOutput plugin should be able to notify other plugins that the capability to transmit packets is unavailable. Likewise, once the route comes back

up, the plugin should notify other plugins that service has been restored. This type of notification would allow plugins earlier in the pipeline to make dynamic routing decisions based on the status of the IP link, such as the re-routing to a non-IP network or performing packet aggregation.

The SmartNet includes an inter-plugin communication framework allowing efficient passing of arbitrary messages between plugins. The framework supports two communication models: publish-subscribe and request-response. In the publish-subscribe model, a plugin registers named objects with the central publish-subscribe subsystem. Once registered, additional plugins may subscribe to the objects, indicating that they are interested in changes to the object. When the originating plugin modifies the object, all subscribed plugins are notified. Notification occurs asynchronously through a message queue that operates in a similar manner to each plugin's packet queue. In the above example, the IP plugin would register a link-state object which indicates whether the link is up or down. Plugins that forward packets to the IP plugin would then subscribe to the link-state object. If they are notified that the link is down, they may choose to route the packet to an alternative destination.

The publish-subscribe model works well for information that aids a performance-based decision, or for information that may be updated frequently. Some information used by plugins may be required before more packet processing can continue, thus waiting for asynchronous notification using the publish-subscribe mechanism would require the plugin to cease packet processing while waiting for an update. In this case, the request-response model is a better fit.

In the request-response model, a plugin can request information directly from another plugin. The requesting plugin would wait for a response before continuing packet processing. For example, suppose the SmartNet contains a routing plugin which provides an interface to the kernel routing tables. The routing plugin could use the publish-subscribe model providing routing updates periodically to all subscribed plugins; however, this would require the routing plugin to continuously poll the kernel routing tables. In addition, if the routing tables were large, this could cause unnecessary communication for routes that may not be used often. Instead, the routing plugin could use the request-response model. In this case, when a plugin is processing a packet, it may send a request directly to the routing plugin.

More importantly, the request can indicate what route is required, thereby eliminating the need to transmit the entire routing table between plugins. The interaction between plugins and the components of the SmartNet core is depicted in Figure 3.2.

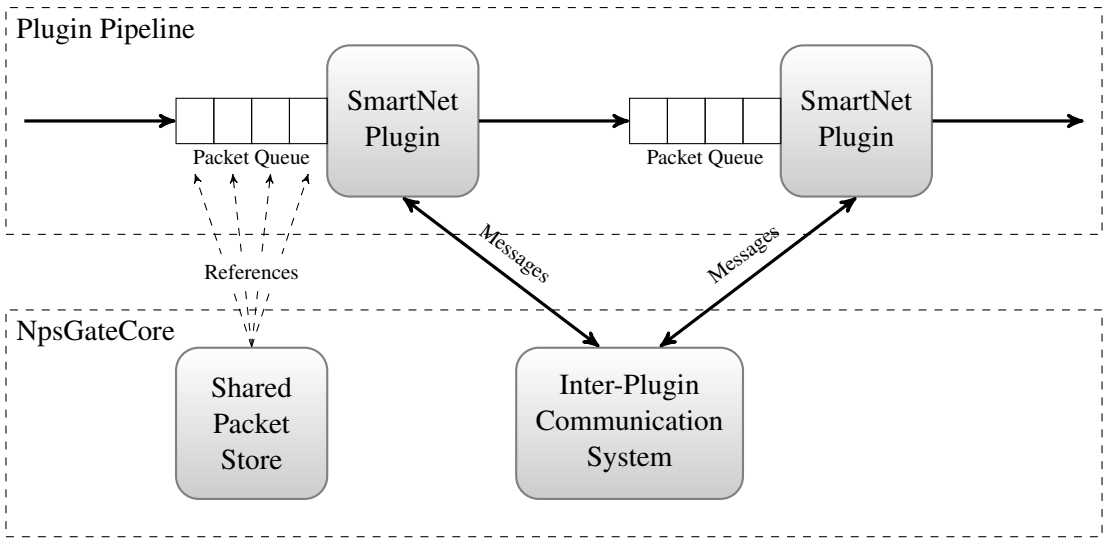


Figure 3.2: SmartNet plugin pipeline. Each plugin contains an input queue with references to packets waiting to be processed. The inter-plugin communication system allows plugins to communicate with each other.

3.2.4 Parallel Processing

The SmartNet architecture regards the packet as the fundamental data unit, thereby supporting parallel processing of individual packets. This gives SmartNet significant advantages in two ways. First, it allows for maximum performance on modern commodity hardware which often has more than one processing unit. Utilizing all hardware processing units can significantly increase the processing speed resulting in lower latencies and higher throughput. Second, since the SmartNet does not know how long processing a single packet might take (due to disruption or other abnormal network conditions), it is important to allow packets traversing a non-disrupted path to proceed while packets over the disrupted path are held.

The concept of individual packet processing is often seen as rudimentary compared to more advanced groupings such as per flow processing; however, due to SmartNet's extensible plugin architecture packet, pipelines can be created to process packets in any grouping desired.

3.3 Application Transparent HTTP SmartNet Solution

The SmartNet design described above allows flexible dynamic network processing and routing. This research will use the SmartNet to transport HTTP over a disrupted network in an application transparent way. The solution presented here is specific to HTTP; however, future work could extend these concepts to other TCP-based application layer protocols.

HTTP relies on TCP as its transport layer protocol. A typical TCP stream operates as a single end-to-end connection between the source and destination hosts. Even though there may be several routers between the source and destination, the routers are agnostic to state of the TCP connection. If a TCP packet is lost during transit, it is the responsibility of the sending host to detect and retransmit the lost packet.

This research will attempt to overcome this challenge with two distinct solutions and evaluate the effictiveness of each solution. The first solution is implementing SplitTCP[3] using several SmartNets. The second solution is using the Spindle DTN[15] implementation to route packets over disrupted links.

3.3.1 SplitTCP

In a SplitTCP network, the routers become TCP aware, and each packet is processed in a hop-by-hop manner. When the first router receives a packet from the source host, it immediately sends an acknowledgment back to the source indicating that the packet was received correctly. The router then forwards the packet to the next router and waits for an acknowledgment. If the packet is lost in the second hop, it is the responsibility of the first router to detect and retransmit the packet. This process continues through all routers until the packet finally reaches the destination host.

This solution is novel in that it uses SplitTCP transparently by means of the SmartNet and does not require any changes to the end hosts or application. Rather than requiring all Internet routers to support SplitTCP, the SmartNet solution uses SplitTCP between SmartNet gateways. Figure 3.3 shows a network consisting of two SmartNet gateways between a HTTP client and HTTP server. In this example, there would be a total of three SplitTCP hops from the client to the server.

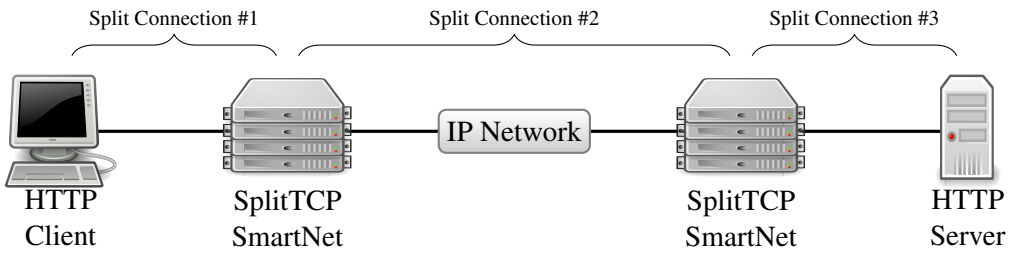


Figure 3.3: End-to-end connection using the SplitTCP plugin. The SplitTCP plugin divides the end-to-end TCP connection in to several hop-by-hop TCP connections.

Using this type of architecture will solve several outstanding issues when operating over a disrupted network. First, since the SmartNet gateways implement SplitTCP, the client and server are effectively only communicating with the SmartNet gateway, meaning they no longer are responsible for ensuring a packet reaches the destination. Second, since SplitTCP generates acknowledgments at every hop, the SmartNet is free to route the packet over multiple networks without affecting either the client or server. Finally, since the SmartNet is aware of possible network disruption, it can adapt to the disruption by delaying packets without causing unnecessary TCP retransmissions.

SplitTCP Plugin Pipeline

The solution presented here is implemented using three SmartNet plugins as shown in Figure 3.4. Each SmartNet gateway will be configured using the same pipeline. The IPInput plugin serves as the only input to the system. It interfaces with the OS to receive packets which need to traverse the SmartNet. In this case, the OS is instructed to deliver all packets with a TCP source or destination port set to 80, the well-known port number for HTTP traffic[16]. All other packets are processed by the OS in its normal fashion. The IPInput plugin performs minimal processing on the raw packet data to transform it to a format suitable for processing by other SmartNet plugins. The IPInput plugin sends all packet to the SplitTCP plugin.

The SplitTCP plugin implements the SplitTCP protocol as described in [3]. Upon receiving a packet, the SplitTCP plugin transmits an acknowledgment to the previous hop, which could be either the sending host or another SmartNet gateway. Next, the SplitTCP plugin

examines packet's TCP state. This state is unique each TCP connection but is also unique to each specific hop. The SplitTCP plugin must maintain a record of each connection's state to associate each packet with a particular TCP connection. Once the packet is associated with a particular connection, the state must be translated to the state for the next hop. Table 3.1 shows an example state table maintained by the SplitTCP plugin. The specific state values shown in Table 3.1 are arbitrary, and are established in the same manner as the TCP state is normally established between end-to-end connections. In the case where a packet can not be associated with a connection, a new entry is created in the state table provided that the packet is part of a TCP three-way handshake. Once the packet is translated, it is forwarded to the Router plugin.

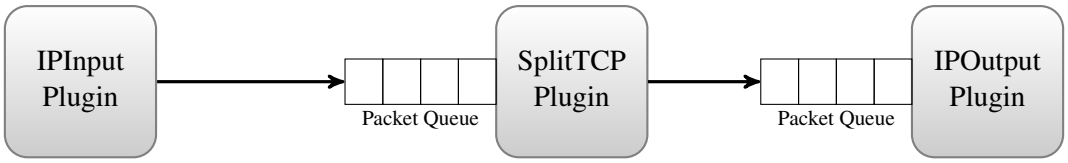


Figure 3.4: SplitTCP plugin pipeline.

The IPOutput plugin is the single exit point from the SmartNet pipeline. It interfaces with the OS or other external software to transfer packets from the SmartNet to be transmitted on the wire. It makes use of the SmartNet inter-plugin communication system to export the state of their respective networks to other plugins.

Source IP:Port	Destination IP:Port	Previous Hop State	Next Hop State
192.168.1.100:37865	10.0.1.1:80	100	8000
10.0.1.1:80	192.168.1.100:37865	30000	2000
192.168.1.101:80	192.168.2.200:45522	400	70000

Table 3.1: SplitTCP plugin state table. The plugin maintains a separate TCP state for both the previous hop and the next hop. As packets arrive, the state is translated to the next hop state.

3.3.2 DTN Solution

In the DTN-based solution, each of the SmartNets shown in Figure 3.5 encapsulates the TCP data packets within the BP. The plugin pipeline configuration, seen in Figure 3.6, consists of two distinct pipelines depending on the source of the packet, either from the IP network, or from the DTN network. Each pipeline essentially takes raw IP packets and

packs them in a bundle and reverses the process for received DTN bundles. Figure 3.6 shows the DTN SmartNet pipeline configuration.

The DTN-based solution has similar benefits to the SplitTCP solution. Both do not require changes to the client or server OS or applications. Each deconstructs a single end-to-end connection into multiple hop-by-hop connections. The DTN solution may provide additional benefits over SplitTCP because the BP was designed for disrupted networks. On the other hand, SplitTCP still uses TCP, which ensures greater compatibility with existing network hardware and does not require the overhead of the overlay architecture used by the DTN solution.

The DTN solution presented here is naïve in a couple of ways. First, this solution simply encapsulates entire IP packets in to a DTN bundle. This increases the total packet size by including not only the encapsulated headers, but also the BP header, and the transport/network layer headers used by the BP. Secondly, the DTN solution does not remove flow control or the reliability mechanisms of TCP. Since the BP provides these functions, the duplicated efforts would decrease performance. Both of these issues will be addressed in Chapter 4 with the specific SmartNet implementation.

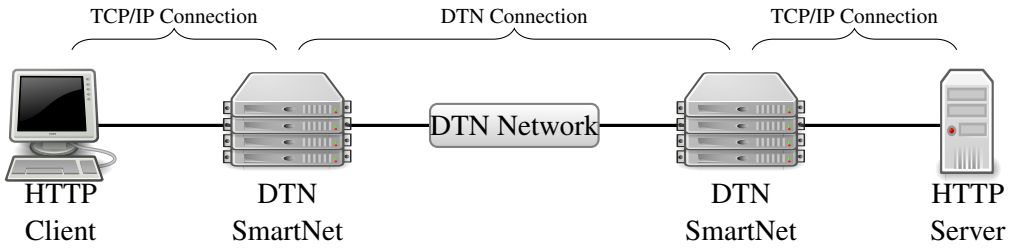


Figure 3.5: End-to-end connection using the DTN plugin. The DTN plugin encapsulates the end-to-end TCP connection within a DTN bundle when traversing between the two SmartNets.

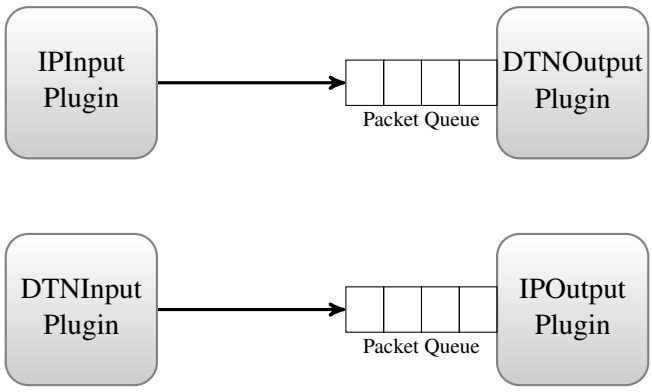


Figure 3.6: DTN plugin pipeline.

CHAPTER 4:
Implementation

4.1 Language and Libraries

NpsGate is implemented from scratch using C++ as the programming language. C++ was chosen because it provides sufficient low-level operations to efficiently manipulate packet-level data structures and hook directly in to the Linux kernel. To avoid duplication of effort, NpsGate relies on several external libraries:

libcrafter (http://code.google.com/p/libcrafter)

Libcrafter is a high-level library designed to make the creation and decoding of network packets easier. It uses C++ classes to provide object-oriented access to IP and TCP headers and payload data. NpsGate uses the libcrafter `Packet` class as the high-level packet abstraction. All plugins receive `Packet` objects and manipulate them using the class interface.

libconfig++ (http://www.hyperrealm.com/libconfig)

Libconfig++ is a C++ library for parsing structured configuration files. The configuration file grammar is simple yet allows for arbitrarily complex configurations. The C++ interface is object-oriented and provides error checking and type-safety. NpsGate uses libconfig++ to parse all plugin configuration files.

Boost C++ libraries (http://www.boost.org)

The Boost is a collection of over 80 individual libraries for the C++ language. Specifically, NpsGate uses the Boost threads, foreach, and heap libraries. Using these libraries increases the portability of NpsGate as well as reducing bugs by using these well-tested third party libraries.

4.2 Class Organization and Description

NpsGate uses an object-oriented approach in its design. Six classes, seen in Figure 4.1, form the NpsGate core which serves as the central coordination mechanism between individual plugins. The six core classes are free to communicate among each other; however,

plugins are only permitted to communicate with the NpsGate core via the `NpsGatePlugin` base class and its associated `PluginCore` class.

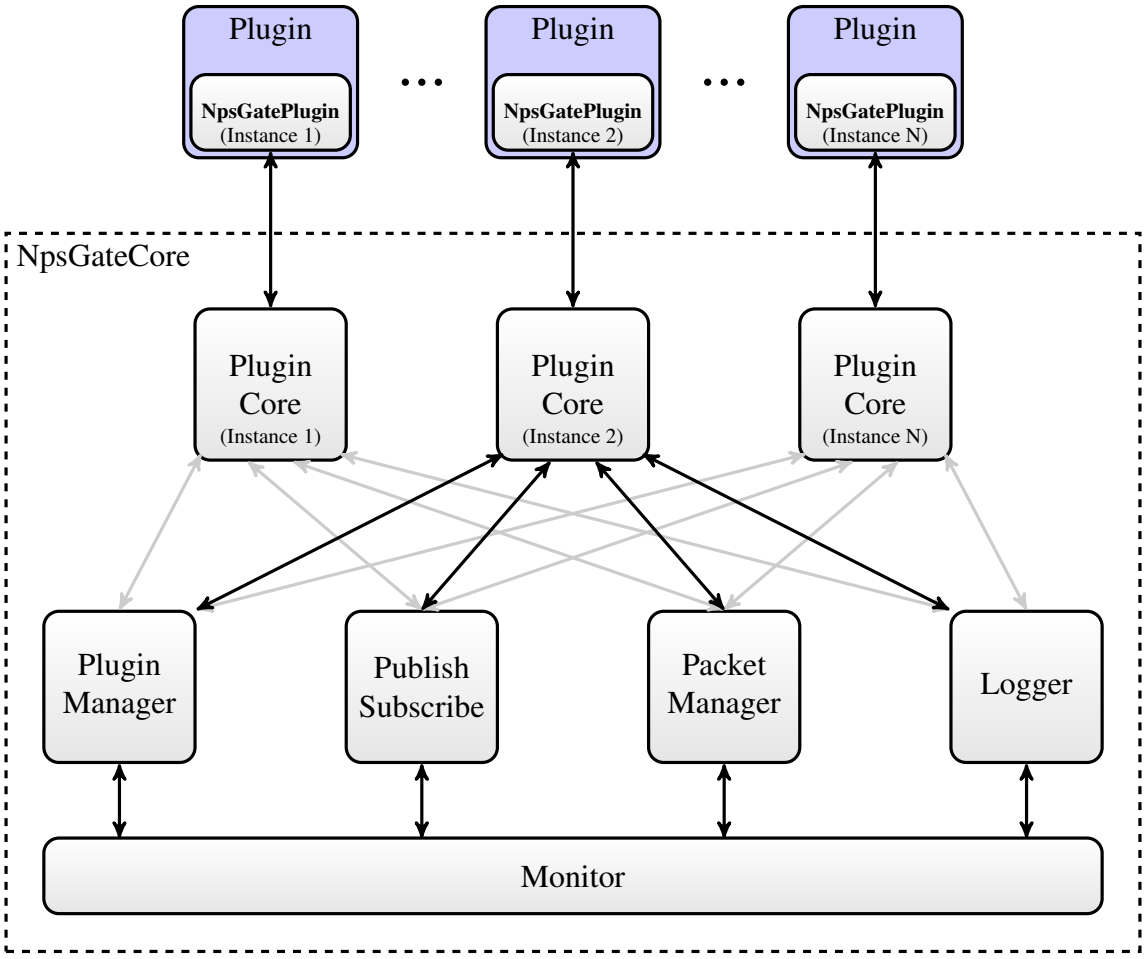


Figure 4.1: NpsGate class diagram. NpsGate is organized in to several classes, each implementing a particular SmartNet feature. The `PluginCore` class serves as the interface between the individual plugins and the NpsGate core.

4.2.1 Plugin Manager

The `PluginManager` class is responsible for the loading, validation, and destruction of individual plugins. Upon starting NpsGate, the `PluginManager` reads the global configuration file and creates a new `PluginCore` instance for each plugin. The `PluginManager` ensures that the shared object contains the appropriate create and destroy functions and then proceeds to spawn a new thread for the plugin and start its initialization routine.

Destruction is handled by sending a cancel signal to the individual plugin's thread. The plugin has the opportunity to perform any cleanup necessary to exit before being unloaded from memory by the `PluginManager`. Plugins that do not respond to the cancel signal in a timely manner are forcibly terminated by sending the SIGKILL signal to the thread.

4.2.2 Plugin Core

During initialization of NpsGate, the `PluginManager` creates an instance of `PluginCore` for each plugin. The `PluginCore` loads the appropriate shared library, parses the plugin configuration file, and ensures the plugin exports the necessary plugin methods. The `PluginCore` instance then spawns a thread and calls the plugin's `main` method.

The `PluginCore` class provides the bridge between a plugin and the NpsGate core. Its interface allows individual plugins to communicate with the NpsGate core and with other plugins. It also provides the thread-safe boundary between the different plugin threads, which ensures that necessary data structures are locked prior to modification. The `NpsPlugin` base class described later is simply a lightweight wrapper around the functionality of the `PluginCore` class.

The `PluginCore` manages the plugin's input queue and performs destination plugin verification when a plugin forwards a packet. The plugin input queue is implemented with the thread-safe `JobQueue` class. The `JobQueue` is a priority queue that stores both `Packet` and `Message` objects. Each type of object is assigned a different priority such that messages always take priority over packets. Since messages are generated as a result of either the publish-subscribe or request-response system, the rationale behind prioritizing messages over packets is so that during the processing of a packet, the plugin will always have the most current information.

Each `PluginCore` object also contains a handle to a `libconfig` configuration file. The configuration is accessed by the plugin using the `Config` object. During initialization of the `PluginCore`, an optional configuration file is read and parsed. Plugins that handle packets are required to have an "outputs" section of the configuration file that lists to where the plugin can forward packets. The `PluginCore` parses these outputs during initialization and restricts the forwarding of packets only to the allowed destinations.

4.2.3 Packet Manager

The `PacketManager` class implements the shared packet store described in Chapter 3. When raw packet data is received by an input plugin, the input plugin creates an instance of the `Packet` class. When the packet is forwarded by the input plugin to another plugin, the `PacketManager` assumes ownership of the packet. The `PacketManager` API is not directly accessible to plugins; instead, plugins interact indirectly through the `PluginCore` class. The packet manager uses a reference counting system to ensure packets are owned by a maximum of one plugin at any given time and that memory is freed once all references to the packet have been released.

The packet manager also provides statistical functions allowing the NpsGate core or other plugins to determine the total number of packets within the SmartNet and the packet throughput. During debugging, the packet manager can provide a step-by-step trace of an individual packet's progression through the plugin pipeline. This functionality can be very useful in determining the flow of packets and identifying any plugins that are "losing" packets.

4.2.4 Publish-Subscribe Subsystem

The `PublishSubscribe` class implements the publish-subscribe subsystem described in Chapter 3. It is responsible for the processing and distribution of all messages between plugins, including memory management of the `Message` objects. The `PublishSubscribe` functionality is encapsulated in three primary methods: `subscribe`, `publish`, and `request`.

The `subscribe` method indicates that the plugin is interested in a particular message. To subscribe, the plugin must provide a fully qualified message name for which it wants to subscribe. Message names consist of the publishing plugin's name followed by an American Standard Code for Information Interchange (ASCII) decimal (.) character, followed by the message name. The subscribe method always succeeds, even if the requested message name has not been previously published. After subscribing, when a message arrives it is placed in the plugin's input queue and is dispatched during the plugin's normal event loop.

The `publish` method is called by a plugin whenever it wishes to transmit information to other interested plugins. The plugin must specify the fully qualified message name along

with a NpsGateVar object. Message names do not need to be registered with the publish-subscribe subsystem prior to publishing; the subsystem will dynamically allocate the new message name and notify any subscribed plugins. The NpsGateVar represents the updated data to publish and is passed as a constant value to all subscribed plugins. Passing as a constant value prevents plugins from changing any of the contents, allowing only a single copy of the NpsGateVar to be shared amongst all plugins.

Message Object

The Message class and the Packet class are the two types of items placed in a plugin's input queue. The Message class encapsulates all components of a inter-plugin message. Each Message has a type, a fully qualified name, and a value parameter. The type, as described in Table 4.1, represents the purpose of the message and how the plugin should handle it.

Message Type	Description
INVALID	Message is invalid and should be ignored (Automatically removed by the NpsGatePlugin base class).
SUBSCRIBE	Message indicating that another plugin has subscribed to something the current plugin published.
SUBSCRIBE_UPDATE	Message received when a variable that was subscribed to has changed.
REQUEST	Message generated when another plugin specifically requests a variable.
REQUEST_UPDATE	Message received when another plugin has responded to a variable request.

Table 4.1: Message types handled by the publish-subscribe subsystem.

The fully qualified name is a string consisting of two components. The first component indicates the plugin to which the message is being sent in the case of a request, or the name of the plugin generating the response in the case of an update or response. The second component indicated a particular variable or action that is of interest. Both components are separated by an ASCII decimal (.). For example, an IPOutput plugin which writes packets to a physical device could provide a variable indicating if the interface is up. The fully qualified name of such a variable might be "IPOutput.link_up".

The value parameter of the Message class is a pointer to a NpsGateVar. The NpsGateVar class is a generic type container that can store any type of data. Example types include strings, integers, floating point numbers, and a list of any of the preceding types. The type stored in the NpsGateVar can be queried at run-time by a plugin and converted to a local variable. In the case of an update or response message, the NpsGateVar parameter indicates the response to the initial request. For requests, the value parameter is optional, but could be used in a plugin specific way to pass additional data with the request.

4.2.5 Logging Subsystem

The logging subsystem, as implemented in the Logger class, provides a consolidated interface for all core modules and plugins to produce run-time logs. It provides five log levels as shown in Table 4.2. Logging information can be written to the standard output, a file, or it can be exported via the Monitor to remote monitoring software. Each log entry contains a timestamp, the plugin generating the event, and the source file name and line number. The NpsGate configuration file includes directives to filter the logging output based on log level and source file name. During production operation it would be typical to only log CRITICAL and WARNING messages.

Log Level	Description
CRITICAL	A critical event such that the SmartNet can not continue operating. Examples include insufficient memory, failing to load a plugin, or the inability to read a configuration file. The SmartNet is halted upon receiving a critical event.
WARNING	An event that is undesirable, but the SmartNet can continue to run. Examples include failing to parse a packet, receiving a message of the wrong data type, or if the OS is dropping packets due to hardware limitations.
INFO	General informative messages used to provide information about the current state of the SmartNet.
DEBUG	More detailed and descriptive messages used to aid in debugging NpsGate or its plugins.
TRACE	Extremely detailed messages. Used to pinpoint specific errors when the debug level doesn't provide sufficient information.

Table 4.2: Log levels provided by the Logger class.

4.2.6 Monitor

The monitor subsystem is designed to export information to external monitoring applications and is implemented as the `Monitor` class. The monitor creates a listen socket on the SmartNet and waits for remote hosts to establish a TCP connection. Once connected, NpsGate and the remote application use a text-based protocol similar to HTTP to transfer information. The following capabilities are supported:

- General statistics about NpsGate including uptime, number of packets processed, and number of bytes processed.
- Reading and making modifications to the NpsGate main configuration file and to individual plugin configuration files.
- Saving configuration files to the local computer.
- Exporting a list of all the plugins along with their pipeline configuration.
- Statistics for each plugin including the number of packets received, packets dropped, and packets forwarded.
- Exporting all log information produced by the `Logger`.
- A list of all the published variables and a list of plugins subscribed to those variables.
- The ability to modify published variables manually and send updates to all subscribed plugins.

4.3 Plugin Interface and Design

While the NpsGate core provides simple yet robust capabilities, the real power of the SmartNet design is in the construction of individual plugins into a complete processing pipeline. The plugin API is designed to be simplistic yet powerful, allowing for unlimited flexibility. When compiled, each plugin is self-contained in its own shared object file that is loaded dynamically at runtime by NpsGate based on the main configuration file. NpsGate provides the `npsgate_plugin.hpp` C++ header file to aid plugin developers.

4.3.1 Plugin Entry Points

NpsGate uses the `dlopen` family of functions to dynamically load plugin shared objects. The `dlsym` function only supports loading functions with C linkage, so each plugin must provide a create and destroy function with C linkage. This is accomplished by

using the NPSGATE_PLUGIN_CREATE and NPSGATE_PLUGIN_DESTROY macros provided in npsgate_plugin.hpp header.

The two macros create the npsgate_create and npsgate_destroy functions with C linkage. The npsgate_create function creates an instance of the plugin's main class (derived from NpsGatePlugin) and returns a pointer to this class. The npsgate_destroy function cleans up and calls the plugin class destructor. These function names are loaded by NpsGate with the dlsym function and are used to create and destroy a plugin instance.

4.3.2 NpsGatePlugin Base Class

All plugins are derived from the NpsGatePlugin base class. Figure 4.2 summarizes the functions contained within the NpsGatePlugin base class. Functions are divided into two categories: virtual functions and processing functions. Virtual functions are overridden by the plugin class and are used as handlers called by NpsGate. Processing functions are provided by NpsGate and are intended to be used by the plugin during processing.

NpsGate Virtual Functions

The six virtual functions specified in the NpsGatePlugin class are described below. Implementation of the first four by each plugin is required for proper operation. The message_timeout function is optional for all plugins and the exit_handler is only required for plugins that do not use the message loop.

`bool init()`

Called by the NpsGate core when the plugin is first loaded. Plugins may perform any initialization required (such as parsing configuration file options); however, it should not rely on other plugins being loaded at this point. Since loading of plugins can occur in any order, it is possible that other plugins have not yet been loaded. Any initialization requiring communication with other plugins should be done in the main function.

`bool main()`

Called by the NpsGate core to start the plugin's main event loop. When this function is called, the plugin is running in its own thread and may begin processing packets and messages. If any initialization requires the use of other plugins, it may be performed in this function since all plugins are guaranteed to be loaded before main

is called. Plugins not serving as an input or output plugin will normally call the `message_loop` function provided by the `NpsGatePlugin` base class to start processing packets. If the `main` function exits, the plugin thread is stopped and it is unloaded from memory.

`bool process_packet(Packet*)`

When using the message loop functionality provided by the `NpsGatePlugin` class, this function is called once for every new packet that arrives in the plugin's input queue. The packet is removed from the input queue and passed as a pointer to the `process_packet` function. Packets belong to only one plugin at a given time so the plugin is able to modify the packet if needed. Once the plugin has finished processing the packet, it should be either forwarded to another plugin using the `forward_packet` function or dropped using the `drop_packet` function. The plugin may hold a reference to the plugin by returning without forwarding or dropping the packet. This is useful in cases where the plugin needs to accumulate multiple packets to make a routing decision. Inside the `process_packet` function packets will continue to accumulate in the input queue; however, the plugin will not receive any notification that new packets have arrived. Therefore, plugins should not perform lengthy operations in this function.

`bool process_message(Message*)`

When using the message loop functionality provided by the `NpsGatePlugin` class, this function is called whenever a new message arrives in the plugin's input queue. The message is removed from the input queue and passed as a pointer to the `process_message` function. Unlike packets, messages may be shared by multiple plugins, so individual plugins are restricted from modifying a `Message`. Additionally, the `Message` object pointer will be invalid once the `process_message` function returns, so plugins needing to a message for longer duration must make a local copy of the message prior to returning.

`bool message_timeout()`

The `message_timeout` function is called whenever packets or messages have not arrived within the time specified using the `set_timeout` function. The timeout functionality provides a method for plugins to perform work at specific intervals while

still using the message loop. This virtual function is optional if the plugins do not need the provided functionality.

`void exit_handler()`

This function is called whenever NpsGate is requesting the plugin to terminate execution. If using the message loop provided by the `NpsGatePlugin` base class, termination of the loop occurs automatically, so no additional processing in this function is required. If the plugin is not using the message loop, this function should signal to the plugin to exit the `main` function as soon as possible. Typically a plugin will have an exit flag that would be set by this function, and subsequently read by the `main` which will trigger a graceful exit. Returning from this function does not immediately terminate the plugin thread so plugin clean up can be deferred to the normal plugin execution flow.

```
class NpsGatePlugin {
    public:
        /* Virtual functions, implemented by plugin developers,
           provide callbacks called by the NpsGate core when
           events occur. */
        virtual bool init();
        virtual bool main();
        virtual bool process_packet(Packet*);
        virtual bool process_message(Message*);
        virtual bool message_timeout();
        virtual void exit_handler();

        /* Member functions of the NpsGatePlugin base class are
           the API used by plugin developers to process packets
           and interact with the rest of the SmartNet. */
        bool forward_packet(string, Packet*);
        bool drop_packet(Packet*);
        bool publish(const string, NpsGateVar*);
        bool subscribe(const string);
        const NpsGateVar* request(string);
        set<string>* get_outputs();
        string get_default_output();
        void message_loop();
        const Config* get_config();
        bool set_timeout(uint32_t);
}
```

Figure 4.2: `NpsGatePlugin` base class. The `NpsGatePlugin` base class defines virtual functions, implemented by each plugin, and processing functions used to interact with the NpsGate core.

NpsGatePlugin Processing Functions

The `NpsGatePlugin` base class provides the following processing functions used to interact with the NpsGate core and other plugins. The `get_config` and `set_timeout` functions may be called at any point after the plugin's `init` function has been called. The remaining functions may only be called after the plugin's `main` function has been called.

`bool forward_packet(string, Packet*)`

Forwards the passed packet to the plugin specified by the first string arguments. The packet pointer can either be a packet removed from the input queue or a packet created by the plugin. For newly created packets, once `forward_packet` has been called, the shared packet store takes custody of the packet and the plugin should not free any memory associated with the packet. After forwarding a packet to another plugin, the local pointer should be invalidated to prevent two plugins from both attempting to modify the packet. The destination plugin name is specified as the first argument. The specified plugin may be any plugin name returned from the `get_outputs` function. Specifying a plugin not in the list of valid output plugins is considered an error and will not be processed by the NpsGate core.

`bool drop_packet(Packet*)`

Drops the specified packet. Dropping the packet removes all references to the packet and it is freed from the shared packet store. Dropping a packet is typically performed by output plugins when the packet has finished traversing the plugin pipeline and has been queued for transmission external to NpsGate. Additionally, plugins that perform significant packet modification, such as aggregating several data packets in to a single packet, should call `drop_packet` on packets that are no longer necessary.

`bool publish(const string, NpsGateVar*)`

Publishes a variable to other subscribed plugins. The first argument is the name of the variable to publish. The NpsGate core will prepend the plugin name to make it fully qualified. The second parameter is a pointer to `NpsGateVar` object that contains the variable data. The publish-subscribe subsystem assumes ownership of the object and will destroy the object once it is no longer needed.

`bool subscribe(const string)`

Subscribes to a variable published by another plugin. The first and only argument is the fully qualified name of the variable to which the plugin wants to subscribe.

When the publishing plugin publishes new data, the message will be inserted in to the calling plugin's input queue. The message loop will remove the message and call the `process_message` function.

`const NpsGateVar* request(const string)`

Sends a request to another plugin for a specific variable. The first argument is the fully qualified name of the variable to request. This function is a blocking function that will not return until a response is generated by the publishing plugin. A pointer to a `NpsGateVar` object, or NULL if the plugin did not satisfy the request, is returned. The `NpsGateVar` object is read-only and the memory is managed by the publish-subscribe subsystem.

`set<string>* get_outputs()`

Returns a list of valid plugins to which the the current plugin can send packets. Valid output are restricted to those specified in the plugin's configuration file. All plugins should verify that the destination plugin is in this list prior to forwarding a packet.

`string get_default_output()`

Returns the default output plugin. This is normally the first output listed in the configuration file and should be used if no specific plugin should receive the packet. In the case of plugins that only have one output, this function can be used exclusively.

`void message_loop`

Starts the message loop provided by the `NpsGatePlugin` base class. The message loop will automatically monitor the input queue for packets and messages, and will call the `process_packet` and `process_message` virtual functions. Additionally, the message loop provides timeout functionality and will gracefully exit when requested by the the NpsGate core. Most plugins that are not input plugins should make use of the message. Input plugins will typically rely on facilities provided by the host OS to implement their own message loop. The call to the `message_loop` function will return when the NpsGate core has signaled that the plugin should exit.

`const Config* get_config()`

Returns a pointer to the plugin's libconfig object. This is the starting point for a plugin to access its own specific configurations options. The `Config` object will only contain the configuration for the current plugin and may not be modified by the plugin.

4.4 Plugins

The initial implementation of NpsGate includes several plugins primarily designed to allow the SmartNet to test the HTTP download scenario discussed in Chapter 1. Each of the plugins uses the NpsGate plugin API described above.

4.4.1 NFQueue

The NFQueue plugin serves as the input plugin for packets that have been received over an IP network. The plugin uses the `libnetfilterqueue` library to hook in to the Linux kernel's NETFILTER subsystem. The NFQueue plugin specifically hooks in to the FORWARD queue, preventing the SmartNet from processing packets destined for the SmartNet host.

NETFILTER is the Linux kernel packet filtering framework. It consists of a set of hooks inside the kernel that allow other kernel modules to register callback functions. The callback functions are called for every packet that traverses the respective hook within the network stack. `libnetfilterqueue` is a user-space library that allows application developers to transfer packets from the NETFILTER subsystem to user-space applications to perform processing, and then re-inject the packet back into the network stack.

During initialization, the NFQueue plugin creates hooks to redirect all incoming packets to the SmartNet. Configuration options can restrict the captured packets to only specific subnets, in effect filtering only certain traffic through the SmartNet. Packets that are not of interest are processed normally by the kernel, therefore they incurr no additional performance penalties. Once a packet is received by NpsGate, the kernel is told to drop the packet and not process it further. Figure 4.3 illustrates the scenario where the NFQueue plugin selectively captures packets for processing by the SmartNet.

When packets are received by the plugin they are formatted as raw data. The first step is to parse and package in to a `Packet` object. Using the `Packet` object increases performance by only requiring parsing of the packet to occur twice; once at input to NpsGate, and once at output, regardless of how many intermediate processing plugins are in the plugin pipeline.

The NFQueue plugin hooks in to NETFILTER in a fault tolerant way. Should NpsGate receive a packet it is unable to process for any reason, or if the NpsGate process is unexpectedly terminated, packets will continue processing through the kernel normally.

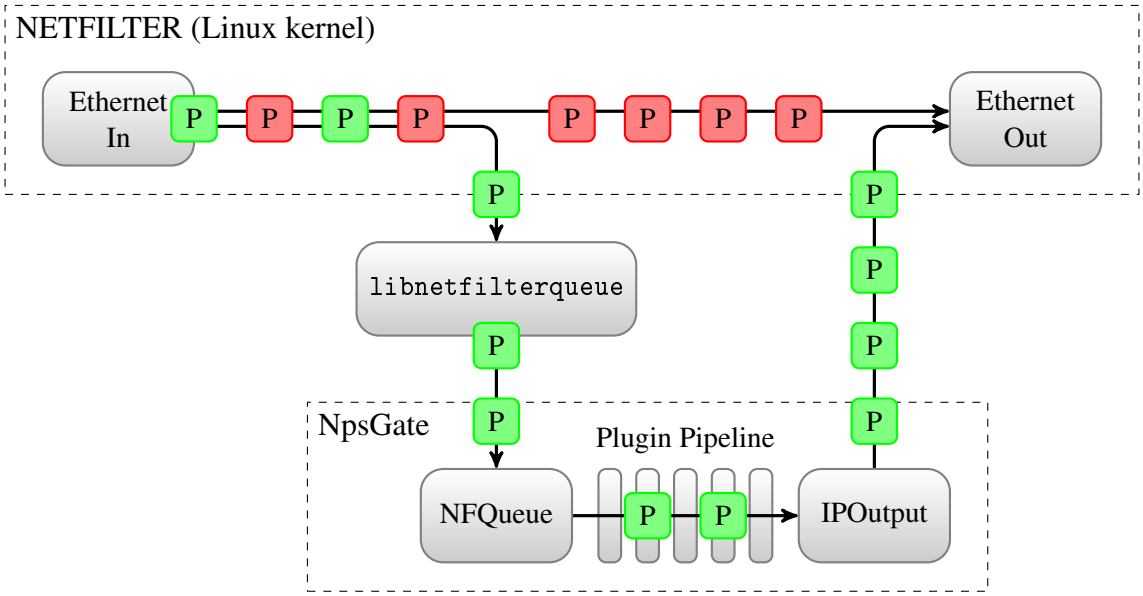


Figure 4.3: NFQueue plugin operation. The NFQueue plugin uses `libnetfilterqueue` to selectively capture packets from the Linux network stack. Packets are processed by the SmartNet, then returned to the OS for transmission.

4.4.2 IPOutput

The IPOutput plugin serves as the primary output plugin and is used when the IP route is up and the network is not in a disrupted state. When the plugin receives a packet in its input queue, it is converted to raw data and pushed to the OS for transmission.

The plugin communicates with the host OS through a single raw socket. The socket is created with the `AF_INET` address family and the `IP_HDRINCL` option is set. The user of an IP level socket rather than using a packet level socket allows NpsGate to use the kernel to prepend the link-layer header along with determining the appropriate link-layer addresses from the kernel's routing tables. The `IP_HDRINCL` option enables NpsGate to fully control the IP header without interference from the host OS.

Since the IPOutput plugin interfaces directly with the host OS, it can query the state of the IP link or the kernel routing tables to determine if a packet is able to reach its destination.

The plugin makes this information available to other plugins in the SmartNet by publishing the link state using the publish-subscribe subsystem. The link state is updated whenever the host OS detects a change in the status of the IP link allowing all plugins in the SmartNet to make dynamic routing decisions.

4.4.3 SplitTCP

The SplitTCP plugin provides the functionality to split an end-to-end TCP connection within the SmartNet similar to what is described in [3]. When a TCP packet is received, the SplitTCP plugin acts as a proxy for the destination host by creating two new TCP endpoints. Each endpoint acts as one of the two communicating hosts, sending acknowledgments to the sending host and taking responsibility for reliable delivery and flow control.

To reduce the amount of new code needed for the plugin, SplitTCP uses the Lightweight Internet Protocol (LWIP) TCP/IP stack to manage the new intermediate TCP endpoints. One endpoint terminates the TCP connection with either the client or server and the other forms a new TCP connection with a peer SplitTCP plugin on the remote SmartNet node. When packets arrive, they are sent to the LWIP stack. LWIP is configured to accept packets for all IP addresses and all TCP ports. LWIP matches the packet with one of the established endpoints and handles the packet by sending acknowledgments for data packets or handling control packets per the TCP standard. When a data packet is received, the data is removed from the packet and placed in the socket queue. Data is read from the socket queue and immediately written to the receiver-side socket. LWIP then generates the necessary TCP packets to send the data to the receiver. The generated packets are then forwarded to the next plugin in the SmartNet pipeline. Figure 4.4 illustrates how the SplitTCP plugin creates two new TCP endpoints and passes the packet data between the endpoints to divide the end-to-end TCP connection. Figure 4.5 shows the effect of using two SplitTCP capable SmartNets on a single end-to-end TCP connection.

The LWIP stack is configurable both at compile time and run time, allowing specific usage scenarios to be fully optimized. Options such as window size, retransmission timeouts and packet aggregation can be adjusted in real-time, to adapt to current network conditions. These optimizations are independent of the end-hosts, requiring no modifications to existing hardware or software configurations. Additionally, using SplitTCP provides the benefit

of hop-by-hop TCP transmission. Multiple SmartNets can be chained together resulting in a single end-to-end TCP connection being composed of several small hops.

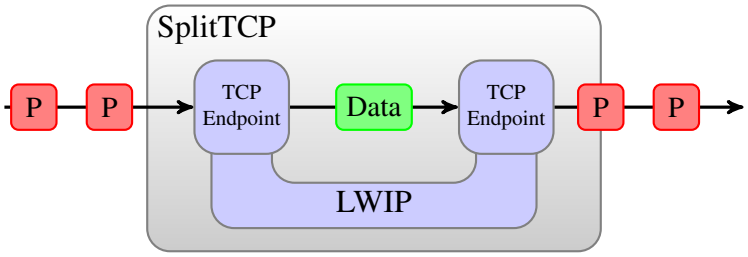


Figure 4.4: SplitTCP plugin operation. The SplitTCP plugin creates two new TCP endpoints. When packets arrive the data is read from one endpoint and sent out from the other endpoint using the LWIP TCP/IP stack.

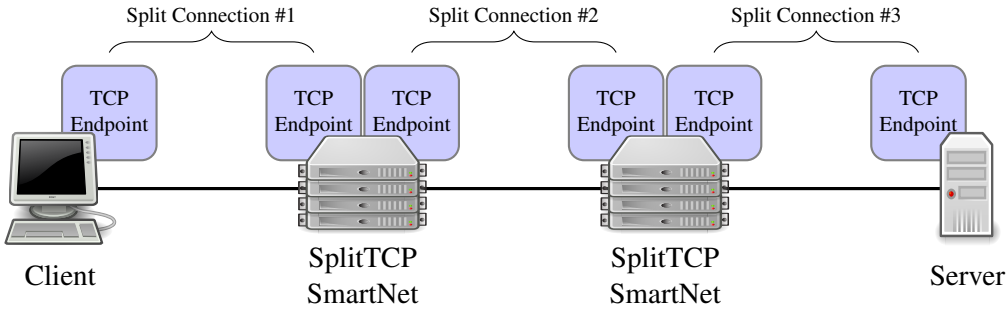


Figure 4.5: End-to-end connection using SplitTCP plugin. The SplitTCP plugin divides the end-to-end TCP connection in to several hop-by-hop TCP connections.

4.4.4 DTNInput

As seen in Figure 4.6, the DTNInput plugin interfaces with a user-space DTN protocol API, receiving packets encapsulated in bundles and forwarding them through the plugin pipeline. The initial version of the DTNInput plugin uses the BBN Spindle bundle protocol agent (BPA) implementation. The plugin registers a DTN unique end-point name within the entire DTN network. The Spindle implementation, which is run as a separate process, will deliver any bundles for the registered end-point to the DTNInput plugin. Once a bundle is received, the data is extracted, parsed into packets, and sent to the next plugin in the pipeline. For the initial implementation, each bundle may contain one and only one IP packet. Future work could add support for multiple packets within each bundle to increase efficiency.

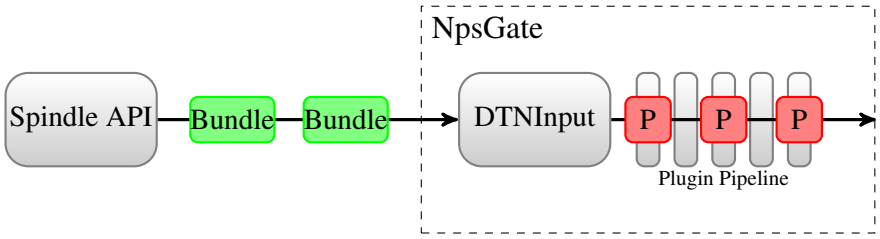


Figure 4.6: DTNInput plugin. The DTNInput plugin receives bundles from the Spindle API. The data from each bundle is extracted and parsed into individual IP packets.

4.4.5 DTNOutput

As seen in Figure 4.7, the DTNOutput plugin allows the SmartNet to send IP packets over a DTN protocol. Like the DTNInput plugin, this plugin uses the BBN Spindle protocol API. Upon initialization, the plugin registers a DTN end-point. All packets received by the plugin are bundled using the Spindle API and sent to the destination end-point. The destination end-point is specified in the plugin configuration file. Each instance of the plugin supports only a single destination end-point name. Multiple instances of the plugin combined with the Router plugin can be used to support multiple DTN destinations.

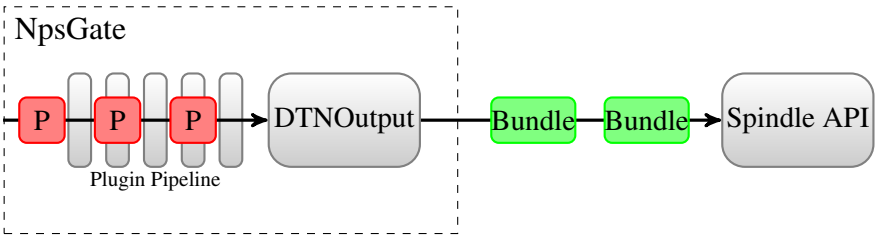


Figure 4.7: DTNOutput plugin. The DTNOutput plugin receives bundles from the plugin pipeline. Each packet is encapsulated in bundles and transmitted via the Spindle API.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Deployment and Testing

In this section, the SmartNet design implemented in Chapter 4 is tested to determine any performance gains or losses incurred running the HTTP download scenario described in Chaper 1. First, the test networks for both the DTN and SplitTCP are described along with enumeration of the different SmartNet configurations used. Next, `iperf` is used to quantify processing overhead caused by the SmartNet and the plugin pipeline in a non-disrupted scenario. The maximum throughput of SmartNet is determined under various network speeds and pipeline lengths. Then both DTN and SplitTCP SmartNet configurations are tested using the HTTP scenario described in Chapter 1. Tests are conducted under different levels of disruption and the various SmartNet configurations are compared. Finally, the flexibility of the SmartNet pipeline is demonstrated by creating a complex pipeline with multiple decision points.

5.1 Deployment Setup

A testbed network was constructed to enable accurate performance measurement of the SmartNet in a controlled environment. The testbed was designed to simulate a typical disrupted network environment with multiple hops between the client and server. Common to each configuration are a HTTP Client, a HTTP Server, and three routers.

The client and server machines were quad-core Intel Core i5 central processing units (CPUs) running at 2.5GHz. Each had 4GB of random access memory (RAM) and a gigabit Ethernet controller. The server ran Ubuntu Linux 13.04 with Apache 2.22 in its default configuration. The client also ran Ubuntu Linux 13.04 using `wget` as the HTTP client. The three routers were low-power dual-core AMD E-350 CPUs running at 1.6GHz. Each had 8GB of RAM and four gigabit Ethernet controllers. The routers ran Vyatta Core 6.6 with the SmartNet implementation and BBN's Spindle DTN software. Each router was capbible of running either as a stand-alone DTN router or as a SmartNet with DTN capibilities.

To test the HTTP download scenario two network layouts were used, one for configurations using DTN and one for configurations using SplitTCP. The two network configurations are described in detail below.

5.1.1 DTN Network Configuration

The network depicted in Figure 5.1 was used to test the performance of the SmartNet in a disrupted environment. Two of the routers were configured to run as SmartNet with the third, central router, configured as a DTN router. The direct links between the SmartNets, HTTP Client, and HTTP Server have a link speed of 1 Gbit/s and suffer no disruption. These links simulate fast and reliable links that typically connect computers on the same local subnet. The links between the two SmartNets and the DTN routers varied in both link speed and disruption pattern. The Linux `tc` utility was used to create a software-based channel emulator allowing fine-grained control of the link speed. Each channel emulator operated in bridged mode to provide minimal impact on the link operation. The disruption pattern was implemented using a custom Perl script on the channel emulator host that set the bridged interfaces up or down. Controlling the link state in this manner on the channel emulator has two key benefits:

1. It allows a deterministic disruption pattern to be implemented and repeated for multiple tests. Each configuration was tested using a pseudo-random disruption pattern. The seeds to the pseudo-random number generator were saved and reused for each subsequent test to ensure that the disruption patterns are consistent between tests.
2. It prevents having to physically disconnect the network cables. Not only is physical disconnection difficult to implement consistently, but most OSs can detect physical disconnection and may prematurely disconnect clients.

The end-to-end link between the two SmartNet gateways is prone to disruption and therefore it is important to ensure both links are DTN capable. The SmartNets support the Spindle DTN protocol via the `DTNInput` and `DTNOutput` plugins, and there is one DTN capable router within the disrupted network. Since one key advantage of DTN is its hop-by-hop transmission, more than one DTN hop is needed to expect any performance benefits relative to TCP.

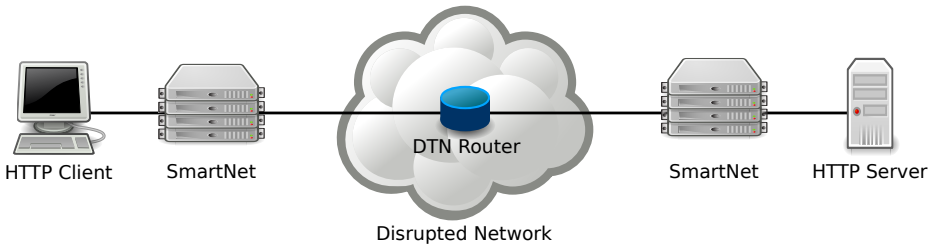


Figure 5.1: DTN-enabled network used to test the SmartNet performance. HTTP performance was tested over a disrupted network traversing two DTN hops with a variety of SmartNet configurations.

5.1.2 SplitTCP Deployment Setup

Testing of a pure SplitTCP-based configuration requires a slightly different network configuration from that of the DTN configuration. This configuration is identical to the DTN Network configuration with the exception that all three routers were configured to run the SmartNet software seen in Figure 5.1. The three SmartNets run SplitTCP over the disrupted connections enabling hop-by-hop transfer of TCP packets. Figure 5.2 illustrates the SplitTCP configuration.

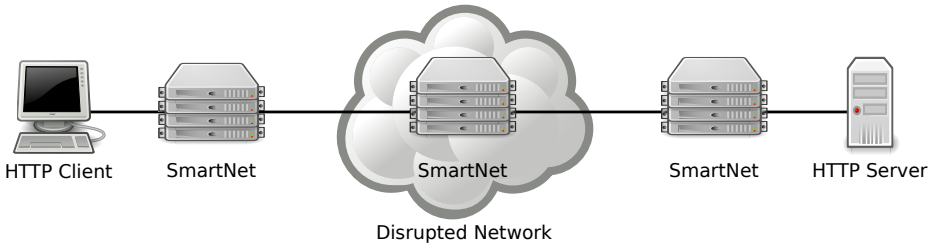


Figure 5.2: SplitTCP network used to test the SmartNet performance. HTTP performance was tested over a disrupted network traversing two SplitTCP hops with a variety of SmartNet configurations.

5.2 SmartNet Overhead

Since SmartNet adds extra processing as each packet traverses the end-to-end connection, it is important to quantify the effect SmartNet has on the overall performance. The overhead of the SmartNet was measured using IPerf[17]. Iperf is a well-known network performance measurement tool often used to optimize network connections. The goal of the Iperf testing was to measure the following two characteristics of the SmartNet:

1. The overhead SmartNet incurs on the host OS by running as a user-space application. Running as a user-space application requires the host OS to transfer packets from kernel-space to user-space. Once SmartNet has completed any processing on the packet, the packet must again be transferred back to kernel-space. Each transfer incurs extra I/O overhead which was measured by comparing the throughput while running SmartNet to the throughput without running SmartNet.
2. The overhead incurred by the plugin pipeline. The SmartNet implementation runs each plugin in its own thread and the packets are passed between plugins using thread-safe queues. The implication of this is that each stage in the pipeline requires a context switch consuming additional processing time. The additional delay caused by the context switches was measured by comparing the SmartNet throughput with increasing pipeline lengths.

Each test consisted of running IPerf in User Datagram Protocol (UDP) mode with various link speeds. Four SmartNet pipeline configurations were tested; one with a pipeline length of two, one with a length of three, one with a length of four, and one with a length of 20 plugins as shown in Figure 5.3. Each pipeline contained a `IPInput` and `IPOutput` plugins with multiple copies of the `Nothing` plugin filling the internal pipeline positions. The `Nothing` plugin is designed to incur minimal overhead by simply passing packets from its input queue to the next plugin without performing any processing. Additionally, a control case was tested without running SmartNet.

The results of the UDP tests are shown in Table 5.1. All SmartNet configurations were able to saturate a 10 Mbit/s connection without packet loss and without any noticeable drop in throughput. At a speed of 30 Mbit/s, performance of the SmartNet started to lag behind the control and Iperf reported packet loss with greater than two plugins. The packet loss at higher speeds can be attributed to SmartNet's use of NFQUEUE to read packets from the OS. NFQUEUE uses a fixed size buffer to transfer packets from the OS to SmartNet. If SmartNet can not read packets fast enough and the buffer becomes full, new packets are dropped.

Link Speed	Packets Transmitted	Pipeline Length	Throughput	Loss
30 Mbit/s	688619	NoSmartNet	30.001 Mbit/s	0.0%
		2	30.001 Mbit/s	0.0%
		3	27.413 Mbit/s	8.5%
		4	27.089 Mbit/s	9.5%
		20	20.621 Mbit/s	30.4%
10 Mbit/s	8504	NoSmartNet	10.001 Mbit/s	0.0%
		2	10.001 Mbit/s	0.0%
		3	10.001 Mbit/s	0.0%
		4	10.001 Mbit/s	0.0%
		20	10.001 Mbit/s	0.0%
1 Mbit/s	852	NoSmartNet	978.460 Kbit/s	0.0%
		2	972.282 Kbit/s	0.0%
		3	972.236 Kbit/s	0.0%
		4	978.170 Kbit/s	0.0%
		20	975.000 Kbit/s	0.0%
512 Kbit/s	437	NoSmartNet	494.249 Kbit/s	0.0%
		2	494.247 Kbit/s	0.0%
		3	494.273 Kbit/s	0.0%
		4	494.363 Kbit/s	0.0%
		20	494.000 Kbit/s	0.0%

Table 5.1: Results from SmartNet overhead testing using IPerf in UDP mode. SmartNet is able to saturate a 10 Mbit/s link without packet loss.

Experiments were made with increasing NFQUEUE buffer size in an effort to prevent packet loss. While this was successful in eliminating packet loss, the processing delay of SmartNet was unable to empty the buffer before `iperf` timed out and terminated the test. Furthermore, increasing the buffer provided only temporary relief while operating at speeds of 30 Mbit/s. Prolonged transfers at this speed would eventually overflow the NFQUEUE buffer again. For this reason, 30 Mbit/s was determined to be the maximum speed for which SmartNet can successfully operate on the low-power hardware used in the testbed.

The second SmartNet measurement was that of pipeline length overhead. Table 5.1 illustrates that the length of the pipeline has negligible impact on the throughput of the SmartNet. Even with a long pipeline of 20 plugins, the throughput drop was less than 0.1% from the control case.

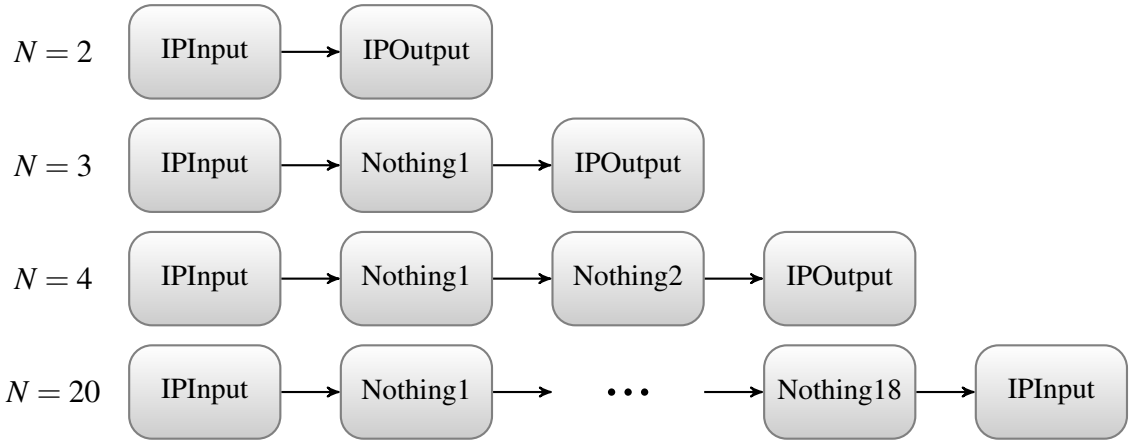


Figure 5.3: SmartNet pipeline length testing configurations. Several SmartNet pipeline lengths (N) were tested to determine what effect context switching between plugins has on throughput.

5.3 HTTP Performance

The overall objective of this research is to improve HTTP performance while operating on a disrupted network by using SmartNet. This section will quantify any performance improvements by testing the SmartNet under various configurations and comparing the performance against a network not running SmartNet. The following configurations were tested:

- ***NoSmartNet:*** A baseline configuration without running SmartNet. Packets flow through the routers using traditional IP forwarding.
- ***IPPassthrough:*** Packets pass through a two-plugin SmartNet pipeline. The IPInput plugin feeds directly in to the IPOutput plugin. No processing is performed on the packets. The pipeline configuration is shown in Figure 5.4.

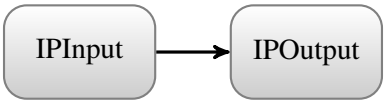


Figure 5.4: IPPassthrough pipeline configuration implemented in each SmartNet gateway.

- ***DTNPassthrough:*** The link between the two SmartNets operate exclusively over DTN. Each SmartNet bundles all received IP packets and likewise unbundle them on the remote side. No additional processing of the IP packets (such as removing TCP flow control or unnecessary ACKs) is performed. The pipeline configuration is shown in Figure 5.5.

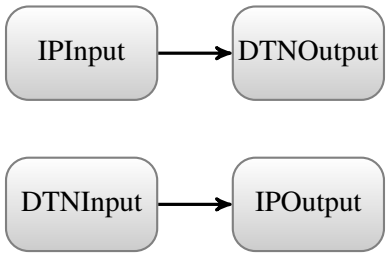


Figure 5.5: DTNPassthrough pipeline configuration implemented in each SmartNet gateway.

- ***DTNBridge:*** Similar to the DTNPassthrough configuration except that the SmartNet bridges the TCP connection over DTN, removing unnecessary elements of the TCP protocol to better optimize over the DTN connection. The SmartNet removes TCP ACKs relying on the DTN protocol to perform reliable delivery and removes TCP congestion and flow control. The pipeline configuration is shown in Figure 5.6.

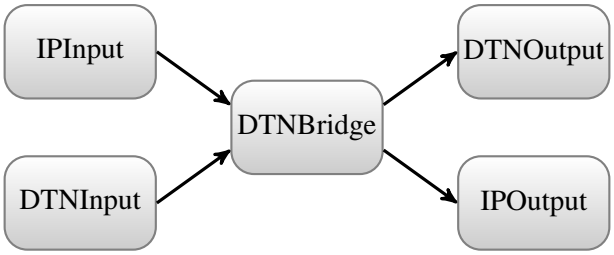


Figure 5.6: DTNBridge pipeline configuration implemented in each SmartNet gateway.

- ***SplitTCP:*** SplitTCP divides a TCP flow at each hop, creating a hop-by-hop TCP connection. In this configuration, there are three SmartNets and the DTN protocol is not used. During a period of disruption, packets only need to be retransmitted over the disrupted link rather than over the entire end-to-end connection. The pipeline configuration is shown in Figure 5.7.

Figure 5.7: SplitTCP pipeline configuration implemented in each SmartNet gateway.

Each configuration was tested on both a non-disrupted network and a disrupted one. The disrupted network was simulated using the channel emulator described in Section 5.1. Each of the two disrupted links was controlled independently using an alternating on-off model. Both the on and off periods were generated from an exponential distribution, with a mean of 45 and 5 seconds, respectively. A graphical depiction of the disruption patterns are illustrated in Figure 5.8.

To evaluate the performance relative to link speed, several different speeds were tested. Link speeds of 30 Mbit/s, 10 Mbit/s, 1 Mbit/s, 512 Kbit/s, 128 Kbit/s, and 64 Kbit/s were selected to fully evaluate both fast and slow links. Additionally, each test was conducted with several file sizes: 10 MB, 1 MB, 100 KB, 10 KB, and 1 KB. The various file sizes allow comparisons for both small and large files, specifically enabling time based features, such as TCP slow start, to be included in the performance analysis.

This work does not aim to directly measure the performance of the DTN protocol or compare various DTN implementations. Rather, measurement of the end-to-end performance while transparently utilizing DTN with traditional TCP clients is the primary goal. To quantify the performance, the primary evaluation metric is the total download time as seen from the client. One test consisted of downloading a single file using a particular link speed and one of the SmartNet configurations. Each test was performed five times in both a disrupted and non-disrupted environment. In total, $1,500$ tests were conducted covering all possible combinations of file size, link speed, SmartNet configuration, and network state.

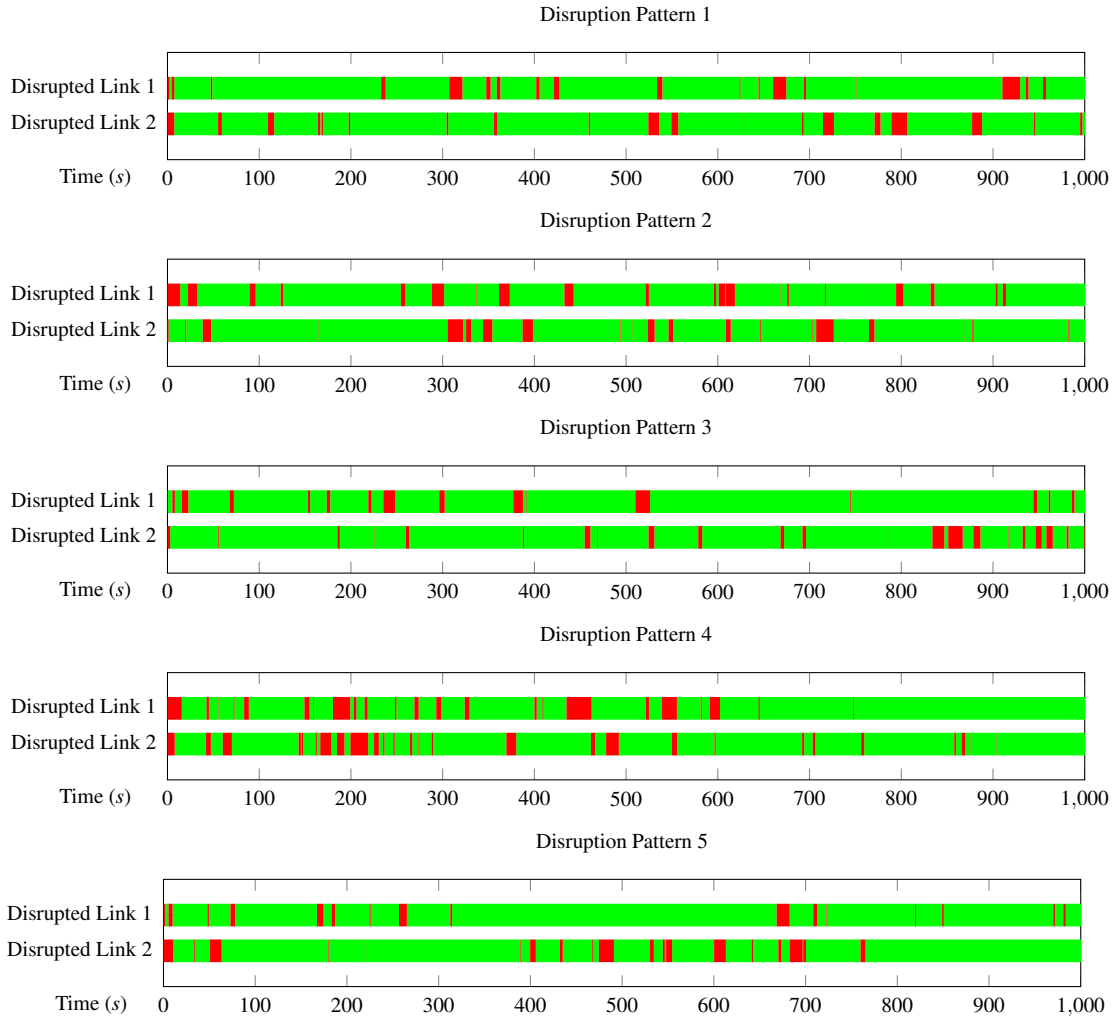


Figure 5.8: Five random disruption patterns used during SmartNet performance testing. Each link was independently disrupted using an exponential distribution. Red indicates periods when the link was down; green indicates periods when the link was up.

Analysis of Performance on a Non-disrupted Network

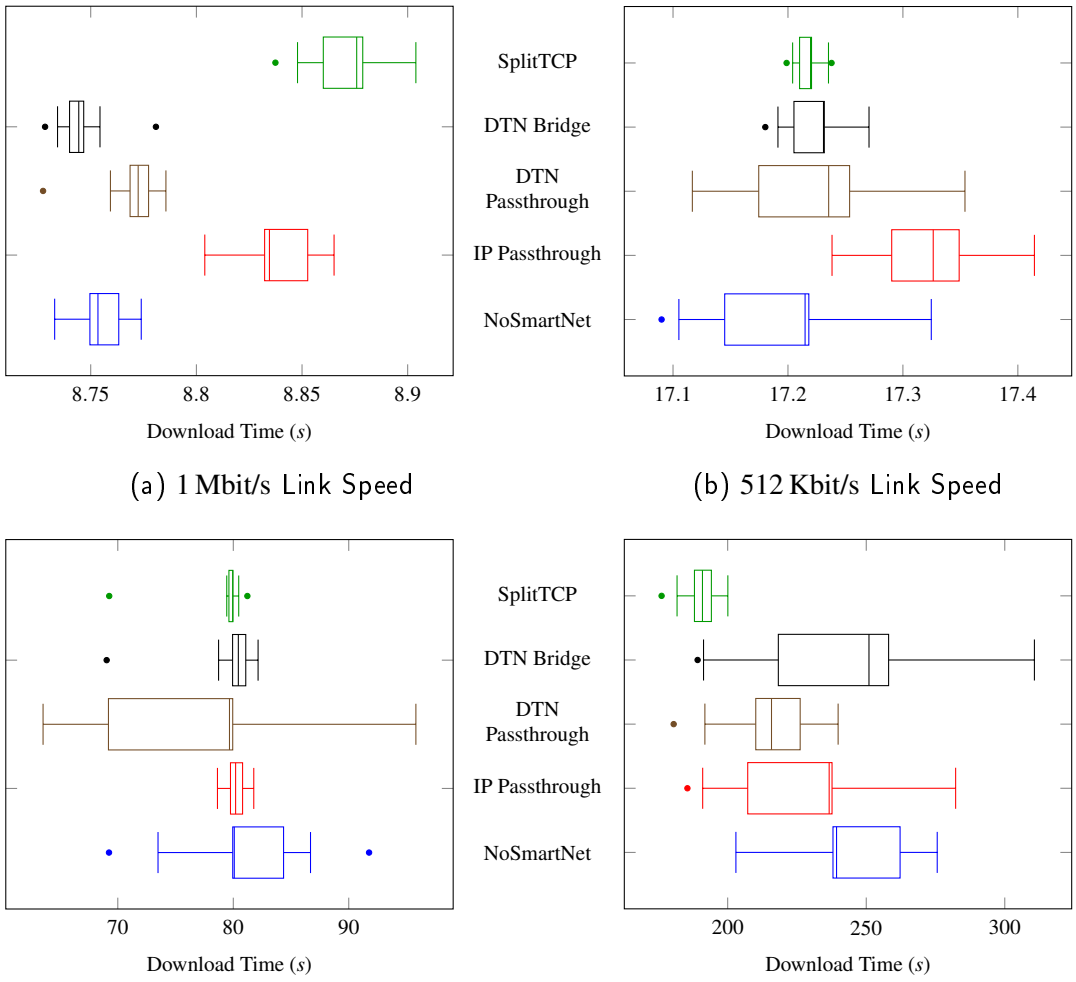
When operating on a non-disrupted network, the average download times of the four SmartNet configurations (DTNPassthough excluded) were within 10% of each other. A box plot of the download times can be seen in Figure 5.9. The box plots depict the download times of the 1 MB file under four link speeds. The download times of the smaller file sizes and under the 10 Mbit/s and 30 Mbit/s links did not clearly identify differences between the con-

figurations because the time to download the files were very short. The complete results of download tests for the non-disrupted network are listed in Appendix A.

With no disruptions, both the NoSmartNet and IPPassthough configurations were capable of operating at their maximum potential, leaving the overhead of the SmartNet itself as the only variable. As expected, the IPPassthough configuration performed only slightly slower than the NoSmartNet configuration, further confirming the SmartNet overhead results in Section 5.2.

In several of the tests, the DTNPassthough configuration performed much slower than the other four configurations. The reasoning for this is that the DTNPassthough simply encapsulates IP packets within a DTN bundle, resulting in the transmission of both the DTN header along with the original TCP and IP headers. Compared with the DTNBridge configuration, which strips both the unnecessary TCP and IP headers along with the removal of TCP flow control, this is a significant disadvantage of using a pure IP-over-DTN model. The DTNBridge configuration performed similarly to both native-IP and IPPassthough, indicating that when properly used, DTN is a viable alternative to TCP even under non-disrupted conditions. While some trends do emerge, it should be noted that the large error bars in Figure 5.9 are a result of only running five tests for each configuration. Additionally, there were several factors not taken in to consideration such as TCP window size, the BP neighbor-discovery timeout, and various BP convergence layer adapterss (CLAs). To fully optimized the SmartNet, all of these additional factors need to be tested to determine the best configuration options for each particular network condition.

The SplitTCP configuration has the potential to increase performance, even on non-disrupted links. The test results show that the SplitTCP download time is always similar to native-IP, but in some cases, decreases the download time. Since TCP speed is limited in the beginning of a transmission by the overall round trip time (RTT) due to its slow start mechanism. By splitting the connection, SplitTCP effectively reduces the RTT calculations to individual segments rather than the whole end-to-end connection, thereby allowing TCP to reach maximum speed at a quicker pace. Additionally, since SplitTCP uses the TCP protocol, it can be seamlessly deployed as a single node on any existing network. Contrast with SmartNets running the BP which must contain at least two nodes, one to bundle packet, and another to unbundle them.



(a) 1 Mbit/s Link Speed

(b) 512 Kbit/s Link Speed

(c) 128 Kbit/s Link Speed

(d) 64 Kbit/s Link Speed

Figure 5.9: Download times of a 1 MB file under various link speeds with no disruption.

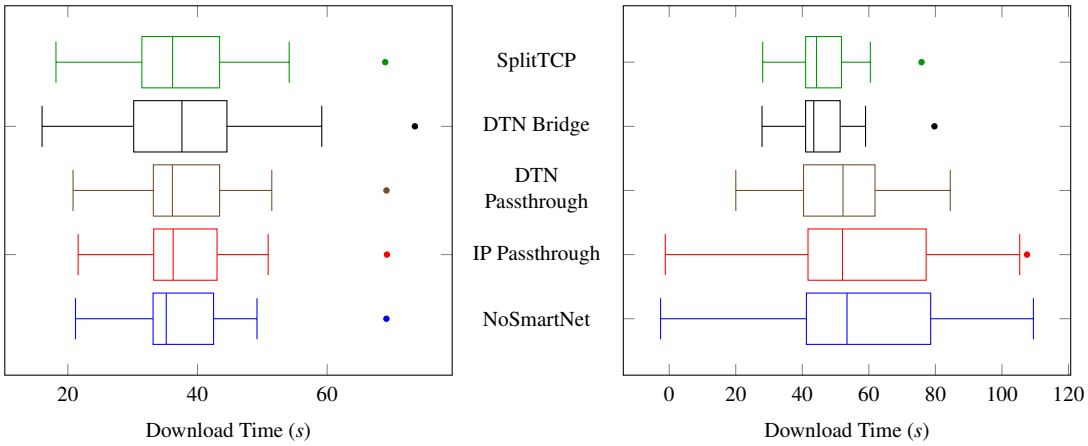
Analysis of Performance on a 10% Disrupted Network

When operating on a 10% disrupted network, there were more significant differences between the five SmartNet configurations than in the non-disrupted case. A box plot of the download times can be seen in Figure 5.10. The box plots depict the download times of the 1 MB file under four link speeds. As in the non-disrupted case, the download times of the smaller file sizes and with the 10 Mbit/s and 30 Mbit/s link speeds did not clearly identify differences between the configurations since the time to download the file was too short to experience a significant number of disruptions. The complete results of download tests for the 10% disrupted network can be found in Appendix B.

Four of the five disruption patterns resulted in similar overall download times, but disruption pattern four proved to be a significant challenge for all the configurations to complete. The download times for pattern four are the outliers in the box plots in Figure 5.10. A closer look at Figure 5.8 indicates that disruption pattern four has a significant amount of disruption near the 200 second mark, resulting in increased download times.

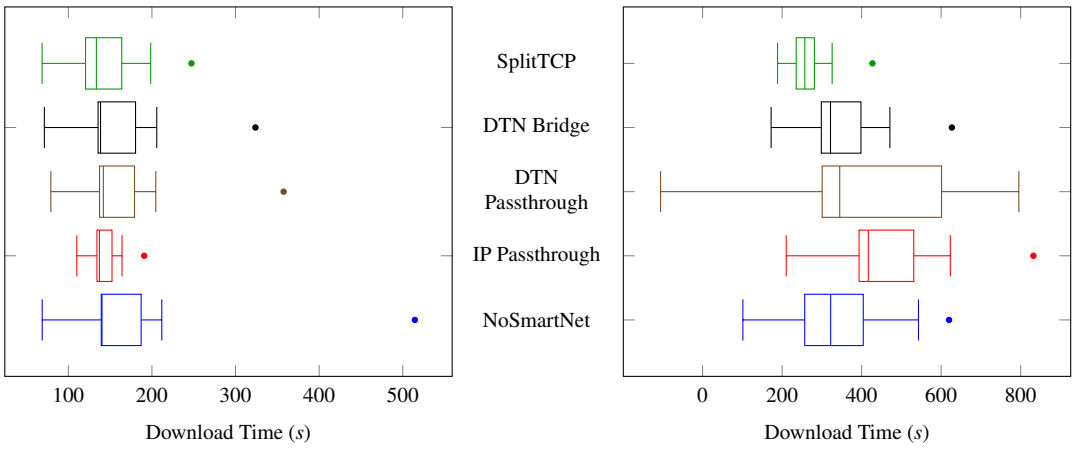
When operating on a 10% disrupted network, both the SplitTCP and DTNBridge configurations typically out performed the control. This result was expected since both configurations use hop-by-hop transfer of packets allowing to move closer to the destination host without the entire end-to-end path being up. Additionally, there tended to be a greater performance increase on the 128 Kbit/s and 64 Kbit/s networks. Since the file sizes were consistent among all the link speed, the slower links required more time to transfer the file and thus was affected more by the network disruptions.

Both the IPPassthrough and DTNPassthrough configurations on average operated more slowly than the control. In the case of the IPPassthrough configuration, this is expected since it does nothing more than pass raw IP packets without performing any processing, while incurring the performance penalty of running through the SmartNet. For DTNPassthrough, raw IP packets are simply encapsulated in the BP. Even though the bundles can traverse hop-by-hop, the TCP flow control and acknowledgments are still handled by the end hosts, taking away all the advantages of using a DTN protocol.



(a) 1 Mbit/s Link Speed

(b) 512 Kbit/s Link Speed



(c) 128 Kbit/s Link Speed

(d) 64 Kbit/s Link Speed

Figure 5.10: Download times of a 10 MB file under various link speeds with 10% disruption.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
Conclusion and Future Work

Flexibility of the SmartNet design was evaluated by constructing several different pipeline configurations and completing 1,500 download tests under multiple network conditions and link speeds. The results from testing have shown that transparently deploying the BP and SplitTCP in existing HTTP applications can increase performance on disrupted networks under certain conditions. In particular, slower link speeds with larger file sizes resulted in a greater number of disruptions and more performance gains by using the BP and SplitTCP. At higher speeds, both solutions performed similar or slightly slower to native-IP.

The SplitTCP solution is lighter weight, requiring less processing power and packet overhead than the BP, and as such tended to outperform the DTN solution in most test cases. Since SplitTCP is backwards compatible with any traditional TCP network hardware, it required no extra protocol overhead unlike the BP which requires additional bundle header information. These results do not dicredit the BP as inferior technology since there are many features unique to the BP that were not tested in this research and may allow the BP to surpass SplitTCP performance in certain situations. For example, one feature of the BP is custody transfer of bundles which includes receipts that are returned back to the sending host. The custody transfer feature would prove useful in situations where a bundle takes a long period of time, on the order of hours or days, to traverse a network. For interactive connections, such as a user downloading information from their web browser, this feature is less useful since the typical user will not wait for lenghtly periods of time for a transfer to complete. In these situations, SplitTCP has the adantage over the DTN solution.

The current SmartNet implementation is primitive in several aspects, yet it lays the ground work for an open and flexible solution for deploying network optimization. The SmartNet concept was developed into a working implementation and tested against a variety of network conditions. Through `iperf` testing, the overhead imposed by the SmartNet system itself was measured and found to have minimal effect on throughput at speeds less than 30 Mbit/s.

Future work should include increased testing of the HTTP download scenario presented in Chapter 1 with additional disruption patterns and file sizes. Specifically, analyzing particular disruption patterns to determine under what specific conditions each possible solution (pure-IP, DTN, or SplitTCP) works best is critical to fully optimizing the connection. Once specific criteria are developed, plugins to detect these specific changes to the network state can be developed. Through these plugins the composable trigger functionality can be used to enable SmartNet to adaptively toggle between DTN, SplitTCP and pure-IP routes depending on the network state.

Within the SmartNet, performance over the DTN link could be improved by aggregating multiple TCP packets in to a single bundle during periods of disruption. This functionality can be performed by an independent plugin, allowing both SplitTCP and the DTN plugins to benefit from aggregating multiple small packets into single large packets.

One of the key design goals of SmartNet is the composition of individual plugins into a series that, when interconnected by queues, forms a packet processing pipeline. To fully realize this goal, an array of new plugins to perform specific network optimizations needs to be developed. Some examples include Network Address Translation, protocol-based packet classification, packet aggregation, and maximum transmission unit (MTU) enforcement.

These plugins would be configured into a complex plugin pipeline forming a mesh or lattice rather than a discrete linear pipeline. Figure 6.1 shows an example of a complex plugin pipeline in a lattice configuration. Decision points within each plugin can cause forks in the pipeline, dynamically sending packets on different paths. Plugin developers can reuse existing publish/subscribe parameters as input to their decision points when appropriate. Merging of two paths is also supported, which allow selected packets to make small deviations for additional processing before rejoining another pipeline. Such a configuration can be customized on a per deployment basis involving a confederation of SmartNet gateways.

In the longer term, additional support for priority-based or proportional scheduling of plugin threads should be considered. The possibility of integrating SmartNet into embedded and real-time OSs would also bring SmartNet's disruption tolerant capabilities to mobile platforms which commonly suffer from disruption. Finally, general code optimization to

increase performance on fast networks and a continuation to develop additional plugins would be emphasized.

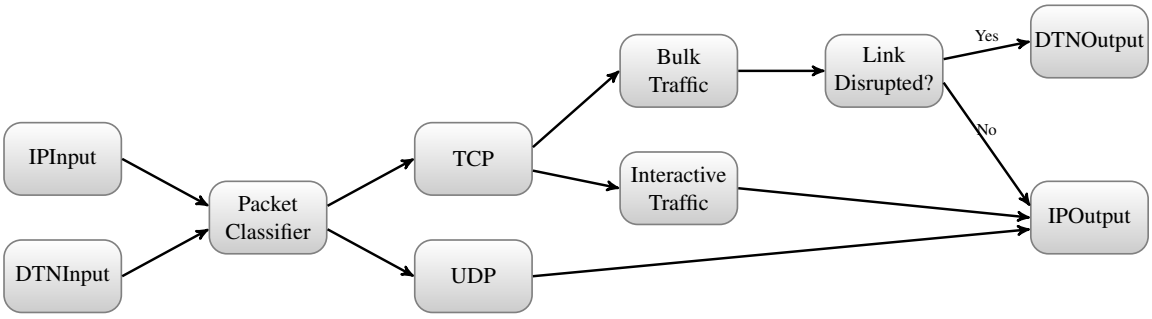


Figure 6.1: An example of a complex processing pipeline. SmartNet is capable of arranging plugins in a mesh or lattice configuration allowing unlimited processing flexibility.

In conclusion, this research has shown our application-transparent solution to current vertical IP/DTN integration approaches may help promote the deployment of DTN technology. The SmartNet solution is novel in that it requires no changes to existing applications and can transparently boost performance on disrupted networks.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
HTTP Download Data (Non-Disrupted Network)

A.1 Link Speed: 30 Mbit/s

Link Speed	File Size	SmartNet Configuration	Download Time (s)				
			1	2	3	4	5
30 Mbit/s	1 KB	NoSmartNet	0.01 –	0.01 –	0.01 –	0.01 –	0.01 –
		IPPassthrough	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%
		DTNPassthrough	0.03 -200.00%	0.03 -200.00%	0.03 -200.00%	0.03 -200.00%	0.03 -200.00%
		DTNBridge	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		SplitTCP	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%
30 Mbit/s	10 KB	NoSmartNet	0.02 –	0.02 –	0.01 –	0.02 –	0.01 –
		IPPassthrough	0.02 0.00%	0.02 0.00%	0.02 -100.00%	0.02 0.00%	0.02 -100.00%
		DTNPassthrough	0.04 -100.00%	0.04 -100.00%	0.04 -300.00%	0.03 -50.00%	0.04 -300.00%
		DTNBridge	0.01 50.00%	0.01 50.00%	0.01 0.00%	0.01 50.00%	0.01 0.00%
		SplitTCP	0.29 $< -1000\%$	0.29 $< -1000\%$	0.29 $< -1000\%$	0.29 $< -1000\%$	0.29 $< -1000\%$
30 Mbit/s	100 KB	NoSmartNet	0.02 –	0.02 –	0.02 –	0.02 –	0.02 –
		IPPassthrough	0.07 -250.00%	0.07 -250.00%	0.07 -250.00%	0.08 -300.00%	0.07 -250.00%
		DTNPassthrough	0.11 -450.00%	0.12 -500.00%	0.12 -500.00%	0.11 -450.00%	0.11 -450.00%
		DTNBridge	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%
		SplitTCP	0.09 -350.00%	0.09 -350.00%	0.35 $< -1000\%$	0.09 -350.00%	0.35 $< -1000\%$
30 Mbit/s	1 MB	NoSmartNet	0.06 –	0.06 –	0.07 –	0.07 –	0.06 –
		IPPassthrough	0.48 -700.00%	0.48 -700.00%	0.48 -585.71%	0.53 -657.14%	0.50 -733.33%
		DTNPassthrough	0.86 $< -1000\%$	0.85 $< -1000\%$	0.84 $< -1000\%$	0.83 $< -1000\%$	0.84 $< -1000\%$
		DTNBridge	0.07 -16.67%	0.06 0.00%	0.07 0.00%	0.06 14.29%	0.06 0.00%
		SplitTCP	0.68 $< -1000\%$	0.66 $< -1000\%$	0.69 -885.71%	0.65 -828.57%	0.64 -966.67%
30 Mbit/s	10 MB	NoSmartNet	0.21 –	0.21 –	0.21 –	0.22 –	0.21 –
		IPPassthrough	4.46 $< -1000\%$	4.65 $< -1000\%$	4.68 $< -1000\%$	4.91 $< -1000\%$	4.74 $< -1000\%$
		DTNPassthrough	8.24 $< -1000\%$	8.09 $< -1000\%$	8.20 $< -1000\%$	8.24 $< -1000\%$	8.19 $< -1000\%$
		DTNBridge	0.22 -4.76%	0.22 -4.76%	0.21 0.00%	0.22 0.00%	0.20 4.76%
		SplitTCP	6.15 $< -1000\%$	6.19 $< -1000\%$	6.06 $< -1000\%$	6.10 $< -1000\%$	6.10 $< -1000\%$

A.2 Link Speed: 10 Mbit/s

Link Speed	File Size	SmartNet Configuration	Download Time (s)				
			1	2	3	4	5
10 Mbit/s	1 KB	NoSmartNet	0.01 –	0.01 –	0.01 –	0.01 –	0.01 –
		IPPassthrough	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.01 0.00%
		DTNPassthrough	0.03 -200.00%	0.02 -100.00%	0.02 -100.00%	0.03 -200.00%	0.03 -200.00%
		DTNBridge	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		SplitTCP	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%
10 Mbit/s	10 KB	NoSmartNet	0.01 –	0.01 –	0.01 –	0.01 –	0.01 –
		IPPassthrough	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%
		DTNPassthrough	0.03 -200.00%	0.03 -200.00%	0.04 -300.00%	0.03 -200.00%	0.04 -300.00%
		DTNBridge	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		SplitTCP	0.29 $< -1000\%$	0.29 $< -1000\%$	0.29 $< -1000\%$	0.29 $< -1000\%$	0.28 $< -1000\%$
10 Mbit/s	100 KB	NoSmartNet	0.02 –	0.02 –	0.02 –	0.02 –	0.02 –
		IPPassthrough	0.07 -250.00%	0.07 -250.00%	0.07 -250.00%	0.07 -250.00%	0.07 -250.00%
		DTNPassthrough	0.11 -450.00%	0.11 -450.00%	0.11 -450.00%	0.12 -500.00%	0.12 -500.00%
		DTNBridge	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%
		SplitTCP	0.09 -350.00%	0.35 $< -1000\%$	0.35 $< -1000\%$	0.35 $< -1000\%$	0.35 $< -1000\%$
10 Mbit/s	1 MB	NoSmartNet	0.06 –	0.06 –	0.06 –	0.06 –	0.05 –
		IPPassthrough	0.49 -716.67%	0.50 -733.33%	0.48 -700.00%	0.49 -716.67%	0.48 -860.00%
		DTNPassthrough	0.83 $< -1000\%$	0.85 $< -1000\%$	0.85 $< -1000\%$	0.85 $< -1000\%$	0.85 $< -1000\%$
		DTNBridge	0.06 0.00%	0.06 0.00%	0.06 0.00%	0.06 0.00%	0.06 -20.00%
		SplitTCP	0.65 -983.33%	0.65 -983.33%	0.63 -950.00%	0.66 $< -1000\%$	0.64 $< -1000\%$
10 Mbit/s	10 MB	NoSmartNet	0.21 –	0.21 –	0.21 –	0.22 –	0.21 –
		IPPassthrough	4.54 $< -1000\%$	4.71 $< -1000\%$	4.47 $< -1000\%$	4.47 $< -1000\%$	4.58 $< -1000\%$
		DTNPassthrough	8.09 $< -1000\%$	8.21 $< -1000\%$	8.26 $< -1000\%$	8.19 $< -1000\%$	8.24 $< -1000\%$
		DTNBridge	0.22 -4.76%	0.22 -4.76%	0.22 -4.76%	0.22 0.00%	0.21 0.00%
		SplitTCP	6.08 $< -1000\%$	6.17 $< -1000\%$	6.09 $< -1000\%$	6.05 $< -1000\%$	6.07 $< -1000\%$

A.3 Link Speed: 1 Mbit/s

Link Speed	File Size	SmartNet Configuration	Download Time (s)				
			1	2	3	4	5
1 Mbit/s	1 KB	NoSmartNet	0.01 –	0.01 –	0.01 –	0.01 –	0.01 –
		IPPassthrough	0.02 -100.00%	0.02 -100.00%	0.01 0.00%	0.02 -100.00%	0.02 -100.00%
		DTNPassthrough	0.02 -100.00%	0.03 -200.00%	0.03 -200.00%	0.03 -200.00%	0.03 -200.00%
		DTNBridge	0.01 0.00%	0.02 -100.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		SplitTCP	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%
1 Mbit/s	10 KB	NoSmartNet	0.08 –	0.09 –	0.09 –	0.09 –	0.08 –
		IPPassthrough	0.09 -12.50%	0.10 -11.11%	0.10 -11.11%	0.10 -11.11%	0.10 -25.00%
		DTNPassthrough	0.13 -62.50%	0.12 -33.33%	0.12 -33.33%	0.12 -33.33%	0.13 -62.50%
		DTNBridge	0.08 0.00%	0.08 11.11%	0.08 11.11%	0.08 11.11%	0.09 -12.50%
		SplitTCP	0.35 -337.50%	0.35 -288.89%	0.35 -288.89%	0.35 -288.89%	0.35 -337.50%
1 Mbit/s	100 KB	NoSmartNet	0.86 –	0.86 –	0.86 –	0.86 –	0.84 –
		IPPassthrough	0.86 0.00%	0.87 -1.16%	0.88 -2.33%	0.87 -1.16%	0.87 -3.57%
		DTNPassthrough	0.95 -10.47%	0.95 -10.47%	0.94 -9.30%	0.95 -10.47%	0.95 -13.10%
		DTNBridge	0.85 1.16%	0.85 1.16%	0.86 0.00%	0.86 0.00%	0.86 -2.38%
		SplitTCP	1.13 -31.40%	1.14 -32.56%	1.15 -33.72%	1.14 -32.56%	1.15 -36.90%
1 Mbit/s	1 MB	NoSmartNet	8.77 –	8.74 –	8.75 –	8.75 –	8.76 –
		IPPassthrough	8.83 -0.68%	8.83 -1.03%	8.83 -0.91%	8.85 -1.14%	8.86 -1.14%
		DTNPassthrough	8.78 -0.11%	8.78 -0.46%	8.73 0.23%	8.77 -0.23%	8.77 -0.11%
		DTNBridge	8.78 -0.11%	8.75 -0.11%	8.74 0.11%	8.74 0.11%	8.73 0.34%
		SplitTCP	8.84 -0.80%	8.88 -1.60%	8.88 -1.49%	8.86 -1.26%	8.88 -1.37%
1 Mbit/s	10 MB	NoSmartNet	87.48 –	87.53 –	87.49 –	87.40 –	87.39 –
		IPPassthrough	87.59 -0.13%	87.35 0.21%	87.55 -0.07%	87.34 0.07%	87.57 -0.21%
		DTNPassthrough	87.50 -0.02%	87.44 0.10%	87.37 0.14%	87.41 -0.01%	87.54 -0.17%
		DTNBridge	87.46 0.02%	87.34 0.22%	87.24 0.29%	87.18 0.25%	87.38 0.01%
		SplitTCP	87.32 0.18%	87.40 0.15%	87.42 0.08%	87.62 -0.25%	87.32 0.08%

A.4 Link Speed: 512 Kbit/s

Link Speed	File Size	SmartNet Configuration	Download Time (s)				
			1	2	3	4	5
512 Kbit/s	1 KB	NoSmartNet	0.01 –	0.01 –	0.01 –	0.01 –	0.01 –
		IPPassthrough	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%
		DTNPassthrough	0.03 -200.00%	0.03 -200.00%	0.03 -200.00%	0.03 -200.00%	0.03 -200.00%
		DTNBridge	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		SplitTCP	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%
512 Kbit/s	10 KB	NoSmartNet	0.15 –	0.15 –	0.16 –	0.16 –	0.15 –
		IPPassthrough	0.19 -26.67%	0.16 -6.67%	0.19 -18.75%	0.18 -12.50%	0.16 -6.67%
		DTNPassthrough	0.21 -40.00%	0.21 -40.00%	0.21 -31.25%	0.23 -43.75%	0.21 -40.00%
		DTNBridge	0.15 0.00%	0.16 -6.67%	0.15 6.25%	0.15 6.25%	0.15 0.00%
		SplitTCP	0.42 -180.00%	0.42 -180.00%	0.42 -162.50%	0.42 -162.50%	0.42 -180.00%
512 Kbit/s	100 KB	NoSmartNet	1.66 –	1.68 –	1.69 –	1.64 –	1.68 –
		IPPassthrough	1.70 -2.41%	1.70 -1.19%	1.70 -0.59%	1.70 -3.66%	1.70 -1.19%
		DTNPassthrough	1.82 -9.64%	1.83 -8.93%	1.83 -8.28%	1.82 -10.98%	1.83 -8.93%
		DTNBridge	1.67 -0.60%	1.68 0.00%	1.66 1.78%	1.64 0.00%	1.66 1.19%
		SplitTCP	1.96 -18.07%	1.96 -16.67%	1.96 -15.98%	1.96 -19.51%	1.95 -16.07%
512 Kbit/s	1 MB	NoSmartNet	17.21 –	17.24 –	17.22 –	17.14 –	17.09 –
		IPPassthrough	17.26 -0.29%	17.29 -0.29%	17.35 -0.75%	17.37 -1.34%	17.33 -1.40%
		DTNPassthrough	17.24 -0.17%	17.14 0.58%	17.25 -0.17%	17.25 -0.64%	17.17 -0.47%
		DTNBridge	17.18 0.17%	17.23 0.06%	17.23 -0.06%	17.21 -0.41%	17.25 -0.94%
		SplitTCP	17.22 -0.06%	17.21 0.17%	17.24 -0.12%	17.22 -0.47%	17.20 -0.64%
512 Kbit/s	10 MB	NoSmartNet	186.03 –	181.17 –	197.07 –	202.27 –	185.22 –
		IPPassthrough	184.59 0.77%	197.38 -8.95%	187.15 5.03%	198.15 2.04%	196.10 -5.87%
		DTNPassthrough	206.11 -10.79%	210.40 -16.13%	211.76 -7.45%	217.77 -7.66%	215.67 -16.44%
		DTNBridge	190.07 -2.17%	182.63 -0.81%	196.72 0.18%	202.64 -0.18%	183.93 0.70%
		SplitTCP	185.98 0.03%	181.93 -0.42%	185.94 5.65%	181.34 10.35%	184.75 0.25%

A.5 Link Speed: 128 Kbit/s

Link Speed	File Size	SmartNet Configuration	Download Time (s)				
			1	2	3	4	5
128 Kbit/s	1 KB	NoSmartNet	0.02 –	0.02 –	0.02 –	0.02 –	0.02 –
		IPPassthrough	0.01 50.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%
		DTNPassthrough	0.04 -100.00%	0.04 -100.00%	0.03 -50.00%	0.05 -150.00%	0.05 -150.00%
		DTNBridge	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%
		SplitTCP	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%
128 Kbit/s	10 KB	NoSmartNet	0.59 –	0.58 –	0.59 –	0.59 –	0.59 –
		IPPassthrough	0.60 -1.69%	0.69 -18.97%	0.59 0.00%	0.59 0.00%	0.59 0.00%
		DTNPassthrough	0.85 -44.07%	0.84 -44.83%	0.84 -42.37%	0.84 -42.37%	0.85 -44.07%
		DTNBridge	0.59 0.00%	0.59 -1.72%	0.58 1.69%	0.58 1.69%	0.59 0.00%
		SplitTCP	0.85 -44.07%	0.85 -46.55%	0.85 -44.07%	0.84 -42.37%	0.85 -44.07%
128 Kbit/s	100 KB	NoSmartNet	6.62 –	6.69 –	6.70 –	6.69 –	6.70 –
		IPPassthrough	6.74 -1.81%	6.72 -0.45%	6.72 -0.30%	6.71 -0.30%	6.74 -0.60%
		DTNPassthrough	7.30 -10.27%	7.31 -9.27%	7.30 -8.96%	7.32 -9.42%	7.30 -8.96%
		DTNBridge	6.70 -1.21%	6.61 1.20%	6.72 -0.30%	6.73 -0.60%	6.61 1.34%
		SplitTCP	6.91 -4.38%	6.88 -2.84%	6.89 -2.84%	6.85 -2.39%	6.88 -2.69%
128 Kbit/s	1 MB	NoSmartNet	69.24 –	79.95 –	84.36 –	91.76 –	80.09 –
		IPPassthrough	81.10 -17.13%	79.76 0.24%	80.20 4.93%	79.43 13.44%	80.81 -0.90%
		DTNPassthrough	92.26 -33.25%	69.18 13.47%	79.95 5.23%	79.67 13.18%	69.19 13.61%
		DTNBridge	69.04 0.29%	80.44 -0.61%	79.94 5.24%	81.08 11.64%	81.37 -1.60%
		SplitTCP	69.26 -0.03%	79.62 0.41%	79.95 5.23%	81.22 11.49%	79.97 0.15%
128 Kbit/s	10 MB	NoSmartNet	869.02 –	861.00 –	879.70 –	877.27 –	857.69 –
		IPPassthrough	983.15 -13.13%	932.15 -8.26%	937.34 -6.55%	940.20 -7.17%	931.93 -8.66%
		DTNPassthrough	1070.49 -23.18%	1066.18 -23.83%	1036.88 -17.87%	1107.74 -26.27%	1070.18 -24.77%
		DTNBridge	862.39 0.76%	859.71 0.15%	879.37 0.04%	856.02 2.42%	865.61 -0.92%
		SplitTCP	858.74 1.18%	859.44 0.18%	867.60 1.38%	864.93 1.41%	892.12 -4.01%

A.6 Link Speed: 64 Kbit/s

Link Speed	File Size	SmartNet Configuration	Download Time (s)				
			1	2	3	4	5
64 Kbit/s	1 KB	NoSmartNet	0.02 –	15.04 –	0.03 –	0.02 –	0.02 –
		IPPassthrough	0.03 -50.00%	0.03 99.80%	0.04 -33.33%	3.03 $< -1000\%$	0.03 -50.00%
		DTNPassthrough	0.05 -150.00%	0.20 98.67%	3.05 $< -1000\%$	3.19 $< -1000\%$	0.06 -200.00%
		DTNBridge	0.02 0.00%	0.02 99.87%	0.03 0.00%	0.02 0.00%	0.02 0.00%
		SplitTCP	6.25 $< -1000\%$	0.02 99.87%	0.02 33.33%	0.02 0.00%	0.02 0.00%
64 Kbit/s	10 KB	NoSmartNet	1.17 –	1.17 –	1.17 –	1.17 –	1.18 –
		IPPassthrough	1.18 -0.85%	1.37 -17.09%	1.37 -17.09%	1.18 -0.85%	1.17 0.85%
		DTNPassthrough	1.65 -41.03%	1.74 -48.72%	1.67 -42.74%	1.85 -58.12%	1.81 -53.39%
		DTNBridge	1.17 0.00%	1.17 0.00%	1.16 0.85%	1.16 0.85%	1.17 0.85%
		SplitTCP	1.43 -22.22%	1.43 -22.22%	1.43 -22.22%	1.43 -22.22%	1.43 -21.19%
64 Kbit/s	100 KB	NoSmartNet	13.48 –	13.49 –	64.97 –	59.11 –	66.49 –
		IPPassthrough	28.41 -110.76%	66.91 -396.00%	13.54 79.16%	77.72 -31.48%	13.49 79.71%
		DTNPassthrough	14.73 -9.27%	15.19 -12.60%	14.75 77.30%	15.28 74.15%	15.32 76.96%
		DTNBridge	103.27 -666.10%	61.68 -357.23%	86.77 -33.55%	76.81 -29.94%	13.48 79.73%
		SplitTCP	13.50 -0.15%	16.32 -20.98%	16.25 74.99%	16.21 72.58%	16.33 75.44%
64 Kbit/s	1 MB	NoSmartNet	238.00 –	262.22 –	239.29 –	228.82 –	263.42 –
		IPPassthrough	185.46 22.08%	236.62 9.76%	207.23 13.40%	242.60 -6.02%	237.66 9.78%
		DTNPassthrough	180.50 24.16%	215.81 17.70%	229.19 4.22%	226.16 1.16%	210.13 20.23%
		DTNBridge	268.74 -12.92%	218.27 16.76%	258.08 -7.85%	250.99 -9.69%	189.14 28.20%
		SplitTCP	199.00 16.39%	176.14 32.83%	194.11 18.88%	190.89 16.58%	188.00 28.63%
64 Kbit/s	10 MB	NoSmartNet	1860.89 –	1913.59 –	1901.71 –	1885.84 –	1948.62 –
		IPPassthrough	1956.87 -5.16%	1915.91 -0.12%	1871.50 1.59%	1872.14 0.73%	1908.05 2.08%
		DTNPassthrough	1981.34 -6.47%	2034.32 -6.31%	1884.99 0.88%	2067.30 -9.62%	1923.15 1.31%
		DTNBridge	1936.04 -4.04%	1878.44 1.84%	1947.66 -2.42%	1935.77 -2.65%	1858.30 4.64%
		SplitTCP	1379.81 25.85%	1373.91 28.20%	1382.33 27.31%	1375.53 27.06%	1374.04 29.49%

APPENDIX B:
HTTP Download Data (10% Disrupted Network)

B.1 Link Speed: 30 Mbit/s

Link Speed	File Size	SmartNet Configuration	Download Time (s)				
			1	2	3	4	5
30 Mbit/s	1 KB	NoSmartNet	0.01 –	0.01 –	0.01 –	0.01 –	0.01 –
		IPPassthrough	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		DTNPassthrough	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		DTNBridge	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		SplitTCP	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%
30 Mbit/s	10 KB	NoSmartNet	0.01 –	0.01 –	0.01 –	0.01 –	0.01 –
		IPPassthrough	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		DTNPassthrough	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		DTNBridge	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		SplitTCP	0.29 $< -1000\%$	0.28 $< -1000\%$	0.29 $< -1000\%$	0.29 $< -1000\%$	0.29 $< -1000\%$
30 Mbit/s	100 KB	NoSmartNet	0.02 –	0.02 –	0.02 –	0.02 –	0.02 –
		IPPassthrough	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.01 50.00%	0.02 0.00%
		DTNPassthrough	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%
		DTNBridge	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%
		SplitTCP	0.35 $< -1000\%$	0.35 $< -1000\%$	0.35 $< -1000\%$	0.35 $< -1000\%$	0.35 $< -1000\%$
30 Mbit/s	1 MB	NoSmartNet	0.06 –	0.05 –	0.06 –	0.06 –	0.05 –
		IPPassthrough	0.06 0.00%	0.06 -20.00%	0.06 0.00%	0.06 0.00%	0.06 -20.00%
		DTNPassthrough	0.06 0.00%	0.06 -20.00%	0.06 0.00%	0.06 0.00%	0.06 -20.00%
		DTNBridge	0.06 0.00%	0.06 -20.00%	0.06 0.00%	0.07 -16.67%	0.06 -20.00%
		SplitTCP	0.65 -983.33%	0.67 $< -1000\%$	0.65 -983.33%	0.64 -966.67%	0.62 $< -1000\%$
30 Mbit/s	10 MB	NoSmartNet	0.23 –	0.22 –	0.22 –	0.22 –	0.23 –
		IPPassthrough	25.88 $< -1000\%$	65.03 $< -1000\%$	38.98 $< -1000\%$	31.90 $< -1000\%$	28.00 $< -1000\%$
		DTNPassthrough	0.23 0.00%	0.22 0.00%	0.22 0.00%	0.22 0.00%	0.23 0.00%
		DTNBridge	0.22 4.35%	0.24 -9.09%	0.22 0.00%	0.22 0.00%	0.21 8.70%
		SplitTCP	26.53 $< -1000\%$	63.70 $< -1000\%$	39.68 $< -1000\%$	33.94 $< -1000\%$	27.75 $< -1000\%$

B.2 Link Speed: 10 Mbit/s

Link Speed	File Size	SmartNet Configuration	Download Time (s)				
			1	2	3	4	5
10 Mbit/s	1 KB	NoSmartNet	0.01 –	0.01 –	0.01 –	0.01 –	0.01 –
		IPPassthrough	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.01 0.00%
		DTNPassthrough	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		DTNBridge	31.08 $< -1000\%$	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		SplitTCP	18.25 $< -1000\%$	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%
10 Mbit/s	10 KB	NoSmartNet	0.02 –	0.02 –	0.02 –	0.01 –	0.01 –
		IPPassthrough	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 -100.00%	0.02 -100.00%
		DTNPassthrough	0.01 50.00%	0.01 50.00%	0.01 50.00%	0.01 0.00%	0.01 0.00%
		DTNBridge	0.01 50.00%	0.01 50.00%	0.01 50.00%	0.01 0.00%	0.01 0.00%
		SplitTCP	0.29 $< -1000\%$	0.29 $< -1000\%$	0.29 $< -1000\%$	0.29 $< -1000\%$	0.28 $< -1000\%$
10 Mbit/s	100 KB	NoSmartNet	0.02 –	0.02 –	0.02 –	0.02 –	0.02 –
		IPPassthrough	0.07 -250.00%	0.07 -250.00%	0.07 -250.00%	0.07 -250.00%	0.07 -250.00%
		DTNPassthrough	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%
		DTNBridge	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%
		SplitTCP	0.35 $< -1000\%$	0.35 $< -1000\%$	0.35 $< -1000\%$	0.35 $< -1000\%$	0.35 $< -1000\%$
10 Mbit/s	1 MB	NoSmartNet	0.06 –	0.06 –	0.06 –	0.06 –	0.06 –
		IPPassthrough	0.49 -716.67%	0.48 -700.00%	0.49 -716.67%	0.48 -700.00%	0.49 -716.67%
		DTNPassthrough	0.06 0.00%	0.06 0.00%	0.06 0.00%	0.06 0.00%	0.06 0.00%
		DTNBridge	0.06 0.00%	0.05 16.67%	0.06 0.00%	0.06 0.00%	0.06 0.00%
		SplitTCP	0.67 $< -1000\%$	0.64 -966.67%	0.66 $< -1000\%$	0.66 $< -1000\%$	0.64 -966.67%
10 Mbit/s	10 MB	NoSmartNet	0.21 –	0.23 –	0.22 –	0.21 –	0.23 –
		IPPassthrough	26.00 $< -1000\%$	64.88 $< -1000\%$	38.76 $< -1000\%$	31.92 $< -1000\%$	28.85 $< -1000\%$
		DTNPassthrough	0.22 -4.76%	0.22 4.35%	0.23 -4.55%	0.21 0.00%	0.22 4.35%
		DTNBridge	0.23 -9.52%	0.22 4.35%	0.23 -4.55%	0.22 -4.76%	0.23 0.00%
		SplitTCP	26.22 $< -1000\%$	65.74 $< -1000\%$	40.01 $< -1000\%$	33.62 $< -1000\%$	27.91 $< -1000\%$

B.3 Link Speed: 1 Mbit/s

Link Speed	File Size	SmartNet Configuration	Download Time (s)				
			1	2	3	4	5
1 Mbit/s	1 KB	NoSmartNet	0.01 –	0.01 –	0.01 –	0.01 –	0.01 –
		IPPassthrough	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		DTNPassthrough	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		DTNBridge	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		SplitTCP	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%
1 Mbit/s	10 KB	NoSmartNet	0.08 –	0.08 –	0.08 –	0.09 –	0.09 –
		IPPassthrough	0.09 -12.50%	0.09 -12.50%	0.09 -12.50%	0.08 11.11%	0.08 11.11%
		DTNPassthrough	0.09 -12.50%	0.08 0.00%	0.09 -12.50%	0.08 11.11%	0.09 0.00%
		DTNBridge	0.09 -12.50%	0.09 -12.50%	0.08 0.00%	0.09 0.00%	0.09 0.00%
		SplitTCP	0.35 -337.50%	0.35 -337.50%	0.35 -337.50%	0.35 -288.89%	0.35 -288.89%
1 Mbit/s	100 KB	NoSmartNet	0.85 –	0.85 –	0.86 –	0.85 –	0.84 –
		IPPassthrough	0.85 0.00%	0.84 1.18%	0.86 0.00%	0.86 -1.18%	0.85 -1.19%
		DTNPassthrough	0.86 -1.18%	0.86 -1.18%	0.86 0.00%	0.85 0.00%	0.85 -1.19%
		DTNBridge	0.86 -1.18%	0.86 -1.18%	0.86 0.00%	0.86 -1.18%	0.84 0.00%
		SplitTCP	1.15 -35.29%	1.15 -35.29%	1.16 -34.88%	1.14 -34.12%	0.89 -5.95%
1 Mbit/s	1 MB	NoSmartNet	33.09 –	69.14 –	42.48 –	35.18 –	33.15 –
		IPPassthrough	29.25 11.60%	69.22 -0.12%	43.01 -1.25%	36.24 -3.01%	33.24 -0.27%
		DTNPassthrough	29.16 11.88%	69.15 -0.01%	43.41 -2.19%	36.13 -2.70%	33.19 -0.12%
		DTNBridge	30.16 8.85%	73.53 -6.35%	44.54 -4.85%	37.59 -6.85%	30.13 9.11%
		SplitTCP	28.16 14.90%	68.95 0.27%	43.41 -2.19%	36.15 -2.76%	31.42 5.22%
1 Mbit/s	10 MB	NoSmartNet	149.83 –	178.45 –	140.75 –	640.71 –	158.55 –
		IPPassthrough	147.32 1.68%	184.12 -3.18%	137.58 2.25%	578.64 9.69%	150.90 4.82%
		DTNPassthrough	147.25 1.72%	181.11 -1.49%	140.99 -0.17%	363.48 43.27%	156.76 1.13%
		DTNBridge	151.41 -1.05%	177.76 0.39%	142.13 -0.98%	394.31 38.46%	150.25 5.23%
		SplitTCP	150.42 -0.39%	183.61 -2.89%	141.31 -0.40%	318.27 50.33%	154.96 2.26%

B.4 Link Speed: 512 Kbit/s

Link Speed	File Size	SmartNet Configuration	Download Time (s)				
			1	2	3	4	5
512 Kbit/s	1 KB	NoSmartNet	0.01 –	0.01 –	0.01 –	0.01 –	0.01 –
		IPPassthrough	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		DTNPassthrough	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		DTNBridge	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%	0.01 0.00%
		SplitTCP	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%	0.02 -100.00%
512 Kbit/s	10 KB	NoSmartNet	0.15 –	0.15 –	0.16 –	0.15 –	0.15 –
		IPPassthrough	0.15 0.00%	0.15 0.00%	0.15 6.25%	0.15 0.00%	0.15 0.00%
		DTNPassthrough	0.15 0.00%	0.15 0.00%	0.16 0.00%	0.16 -6.67%	0.15 0.00%
		DTNBridge	0.15 0.00%	0.15 0.00%	0.15 6.25%	0.15 0.00%	0.15 0.00%
		SplitTCP	0.42 -180.00%	0.42 -180.00%	0.42 -162.50%	0.42 -180.00%	0.42 -180.00%
512 Kbit/s	100 KB	NoSmartNet	1.64 –	1.69 –	1.68 –	1.65 –	1.66 –
		IPPassthrough	1.69 -3.05%	1.69 0.00%	1.63 2.98%	1.68 -1.82%	1.68 -1.20%
		DTNPassthrough	1.68 -2.44%	1.66 1.78%	1.68 0.00%	1.66 -0.61%	1.67 -0.60%
		DTNBridge	1.67 -1.83%	1.66 1.78%	1.69 -0.60%	1.67 -1.21%	1.68 -1.20%
		SplitTCP	1.95 -18.90%	1.95 -15.38%	1.95 -16.07%	1.93 -16.97%	1.96 -18.07%
512 Kbit/s	1 MB	NoSmartNet	38.41 –	78.58 –	53.44 –	84.26 –	41.23 –
		IPPassthrough	38.39 0.05%	77.22 1.73%	52.08 2.54%	107.54 -27.63%	41.71 -1.16%
		DTNPassthrough	36.41 5.21%	77.42 1.48%	52.24 2.25%	61.85 26.60%	40.37 2.09%
		DTNBridge	40.76 -6.12%	79.72 -1.45%	51.37 3.87%	43.45 48.43%	41.00 0.56%
		SplitTCP	36.79 4.22%	75.84 3.49%	51.79 3.09%	44.27 47.46%	41.00 0.56%
512 Kbit/s	10 MB	NoSmartNet	273.43 –	291.17 –	328.05 –	589.26 –	296.52 –
		IPPassthrough	277.56 -1.51%	294.19 -1.04%	341.90 -4.22%	595.48 -1.06%	286.66 3.33%
		DTNPassthrough	284.40 -4.01%	287.52 1.25%	437.93 -33.49%	651.35 -10.54%	293.92 0.88%
		DTNBridge	272.82 0.22%	286.04 1.76%	349.46 -6.53%	D –	263.05 11.29%
		SplitTCP	296.24 -8.34%	281.13 3.45%	335.84 -2.37%	579.00 1.74%	294.37 0.73%

B.5 Link Speed: 128 Kbit/s

Link Speed	File Size	SmartNet Configuration	Download Time (s)				
			1	2	3	4	5
128 Kbit/s	1 KB	NoSmartNet	63.14 –	1.01 –	0.02 –	0.02 –	0.02 –
		IPPassthrough	1.02 98.38%	0.02 98.02%	0.02 0.00%	0.02 0.00%	0.02 0.00%
		DTNPassthrough	0.02 99.97%	0.02 98.02%	0.02 0.00%	0.02 0.00%	0.01 50.00%
		DTNBridge	0.01 99.98%	0.02 98.02%	0.02 0.00%	0.02 0.00%	0.02 0.00%
		SplitTCP	0.02 99.97%	0.02 98.02%	0.02 0.00%	0.02 0.00%	0.02 0.00%
128 Kbit/s	10 KB	NoSmartNet	0.59 –	0.59 –	0.60 –	0.59 –	0.59 –
		IPPassthrough	0.59 0.00%	0.69 -16.95%	0.59 1.67%	0.59 0.00%	0.68 -15.25%
		DTNPassthrough	0.58 1.69%	0.58 1.69%	0.59 1.67%	0.59 0.00%	0.59 0.00%
		DTNBridge	0.59 0.00%	0.60 -1.69%	0.59 1.67%	0.59 0.00%	0.59 0.00%
		SplitTCP	0.85 -44.07%	0.85 -44.07%	0.84 -40.00%	0.86 -45.76%	0.85 -44.07%
128 Kbit/s	100 KB	NoSmartNet	86.89 –	60.23 –	119.94 –	104.14 –	100.94 –
		IPPassthrough	118.25 -36.09%	62.64 -4.00%	58.81 50.97%	105.95 -1.74%	68.05 32.58%
		DTNPassthrough	49.56 42.96%	59.55 1.13%	55.31 53.89%	63.27 39.25%	61.83 38.75%
		DTNBridge	57.89 33.38%	57.14 5.13%	49.21 58.97%	53.56 48.57%	55.09 45.42%
		SplitTCP	52.93 39.08%	100.92 -67.56%	101.00 15.79%	100.90 3.11%	53.11 47.38%
128 Kbit/s	1 MB	NoSmartNet	139.39 –	187.08 –	140.28 –	514.52 –	129.47 –
		IPPassthrough	135.01 3.14%	190.76 -1.97%	131.89 5.98%	D –	139.41 -7.68%
		DTNPassthrough	141.87 -1.78%	179.18 4.22%	122.92 12.38%	357.51 30.52%	137.35 -6.09%
		DTNBridge	138.55 0.60%	180.41 3.57%	134.91 3.83%	323.81 37.07%	135.60 -4.73%
		SplitTCP	110.69 20.59%	163.91 12.39%	120.60 14.03%	247.25 51.95%	133.55 -3.15%
128 Kbit/s	10 MB	NoSmartNet	1754.22 –	1719.39 –	1881.77 –	1580.12 –	1539.18 –
		IPPassthrough	1688.06 3.77%	1391.78 19.05%	D –	1774.20 -12.28%	1955.36 -27.04%
		DTNPassthrough	1778.46 -1.38%	D –	D –	1560.10 1.27%	1715.91 -11.48%
		DTNBridge	D –	1517.76 11.73%	1751.85 6.90%	D –	1618.40 -5.15%
		SplitTCP	1207.36 31.17%	1182.48 31.23%	1194.85 36.50%	1309.13 17.15%	1144.29 25.66%

B.6 Link Speed: 64 Kbit/s

Link Speed	File Size	SmartNet Configuration	Download Time (s)				
			1	2	3	4	5
64 Kbit/s	1 KB	NoSmartNet	0.02 –	0.02 –	0.02 –	0.02 –	0.02 –
		IPPassthrough	0.03 -50.00%	0.03 -50.00%	0.03 -50.00%	0.03 -50.00%	0.03 -50.00%
		DTNPassthrough	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%
		DTNBridge	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%
		SplitTCP	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%	0.02 0.00%
64 Kbit/s	10 KB	NoSmartNet	1.16 –	1.17 –	1.17 –	1.17 –	1.18 –
		IPPassthrough	1.18 -1.72%	1.18 -0.85%	1.18 -0.85%	1.18 -0.85%	1.37 -16.10%
		DTNPassthrough	1.17 -0.86%	1.17 0.00%	1.17 0.00%	1.17 0.00%	1.17 0.85%
		DTNBridge	1.17 -0.86%	1.17 0.00%	1.17 0.00%	1.17 0.00%	1.17 0.85%
		SplitTCP	1.43 -23.28%	1.43 -22.22%	1.41 -20.51%	1.43 -22.22%	1.43 -21.19%
64 Kbit/s	100 KB	NoSmartNet	42.28 –	902.61 –	163.42 –	143.56 –	94.98 –
		IPPassthrough	36.99 12.51%	34.28 96.20%	57.39 64.88%	40.52 71.77%	36.56 61.51%
		DTNPassthrough	40.89 3.29%	40.92 95.47%	84.85 48.08%	45.49 68.31%	97.16 -2.30%
		DTNBridge	88.34 -108.94%	98.10 89.13%	42.36 74.08%	103.44 27.95%	102.65 -8.08%
		SplitTCP	18.73 55.70%	18.73 97.92%	43.57 73.34%	43.42 69.75%	18.49 80.53%
64 Kbit/s	1 MB	NoSmartNet	404.09 –	256.87 –	322.18 –	619.86 –	254.71 –
		IPPassthrough	402.60 0.37%	D –	430.86 -33.73%	832.10 -34.24%	366.59 -43.92%
		DTNPassthrough	601.12 -48.76%	253.40 1.35%	344.93 -7.06%	663.81 -7.09%	300.73 -18.07%
		DTNBridge	398.24 1.45%	252.73 1.61%	321.73 0.14%	627.04 -1.16%	298.64 -17.25%
		SplitTCP	257.18 36.36%	231.34 9.94%	281.17 12.73%	427.21 31.08%	235.40 7.58%
64 Kbit/s	10 MB	NoSmartNet	4468.18 –	D –	4248.38 –	D –	3324.08 –
		IPPassthrough	D –	4180.59 $< -1000\%$	D –	D –	4072.10 -22.50%
		DTNPassthrough	D –	D –	D –	D –	D –
		DTNBridge	D –	D –	4116.42 3.11%	3653.21 $< -1000\%$	3740.31 -12.52%
		SplitTCP	2470.44 44.71%	2368.78 $< -1000\%$	2488.01 41.44%	2562.17 $< -1000\%$	2343.63 29.50%

REFERENCES

[1] S. Farrell, V. Cahill, D. Geraghty, I. Humphreys, and P. McDonald, "When TCP breaks: delay- and disruption- tolerant networking," *IEEE Internet Computing*, vol. 10, no. 4, pp. 72–78, Jul. 2006. DOI: 10.1109/MIC.2006.91.

[2] S. Burleigh and K. Scott, "Bundle protocol specification," RFC Editor, RFC 5050, Nov. 2007, pp. 1–50. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5050.txt.

[3] S. Kopparty, S. Krishnamurthy, M. Faloutsos, and S. Tripathi, "Split TCP for mobile ad hoc networks," in *Global Telecommunications Conference, 2002.*, vol. 1, Nov. 2002, pp. 138–142. DOI: 10.1109/GLOCOM.2002.1188057.

[4] D. Brown, K. Trinidad, and R. Borja. (2008). NASA successfully tests first deep space internet, NASA, [Online]. Available: http://www.nasa.gov/hqnews/2008/nov/HQ_08-298_Deep_space_internet.html.

[5] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, San Jose, CA: USENIX Association, 2012, pp. 24–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228331.

[6] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, Hong Kong, China: ACM, 2013, pp. 27–38, ISBN: 978-1-4503-2056-6. DOI: 10.1145/2486001.2486022.

[7] M. Demmer, E. Brewer, K. Fall, S. Jain, M. Ho, and R. Patra, "Implementing delay tolerant networking," Intel Research Berkeley, Tech. Rep. IRBTR-04-020, 2004.

[8] J. Martins, M. Ahmed, C. Raiciu, and F. Huici, "Enabling fast, dynamic network processing with clickos," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, Hong Kong, China: ACM, 2013, pp. 67–72. DOI: 10.1145/2491185.2491195.

[9] K. Fall, "A delay-tolerant network architecture for challenged internets," in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Karlsruhe, Germany: ACM, 2003, pp. 27–34. DOI: 10.1145/863955.863960.

[10] J. Rohrer and G. Xie, "DTN hybrid networks for vehicular communications," in *Connected Vehicles and Expo (ICCVE), 2013 International Conference on*, Dec. 2013, pp. 114–120. DOI: 10.1109/ICCVE.2013.6799779.

[11] S. Schildt, J. Morgenroth, W.-B. Pöttner, and L. Wolf, "IBR-DTN: a lightweight, modular and highly portable bundle protocol implementation," *Electronic Communications of the EASST*, vol. 37, 2011.

[12] S. Burleigh, "Interplanetary overlay network: an implementation of the DTN bundle protocol," in *Consumer Communications and Networking Conference, 2007. CCNC 2007*, Jan. 2007, pp. 222–226. DOI: 10.1109/CCNC.2007.51.

[13] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: enabling innovation in middlebox deployment," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, Cambridge, Massachusetts: ACM, 2011, 21:1–21:6. DOI: 10.1145/2070562.2070583.

[14] *Media Access Control (MAC) Bridges and Virtual Bridge Local Area Networks*, IEEE Std 802.1Q, 2011.

[15] R. Krishnan, P. Basu, J. M. Mikkelson, C. Small, R. Ramanathan, D. W. Brown, J. R. Burgess, A. L. Caro, M. Condell, N. C. Goffee, R. R. Hain, R. E. Hansen, C. Jones, V. Kawadia, D. P. Mankins, B. I. Schwartz, W. T. Strayer, J. W. Ward, D. P. Wiggins, and S. H. Polit, "The SPINDLE disruption-tolerant networking system," in *Military Communications Conference, 2007. MILCOM 2007. IEEE*, Oct. 2007, pp. 1–7. DOI: 10.1109/MILCOM.2007.4454942.

[16] *Service name and transport protocol port number registry*, Internet Assigned Numbers Authority (IANA), Feb. 2014. [Online]. Available: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt.

[17] *Iperf*, 2014. [Online]. Available: http://iperf.fr.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California